Titre: Title:	Architecture hybride délibérative/réactive pour le contrôle d'une équipe hétérogène de robots mobiles
Auteur: Author:	Raphaël Gava
Date:	2007
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Gava, R. (2007). Architecture hybride délibérative/réactive pour le contrôle d'une équipe hétérogène de robots mobiles [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8045/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/8045/
Directeurs de recherche: Advisors:	
Programme: Program:	Non spécifié

UNIVERSITÉ DE MONTRÉAL

ARCHITECTURE HYBRIDE DÉLIBÉRATIVE/RÉACTIVE POUR LE CONTRÔLE D'UNE ÉQUIPE HÉTÉROGÈNE DE ROBOTS MOBILES

RAPHAËL GAVA
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
JUILLET 2007

Your file Votre référence
ISBN: 978-0-494-35679-1
Our file Notre référence
ISBN: 978-0-494-35679-1

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

ARCHITECTURE HYBRIDE DÉLIBÉRATIVE/RÉACTIVE POUR LE CONTRÔLE D'UNE ÉQUIPE HÉTÉROGÈNE DE ROBOTS MOBILES

présenté par: GAVA Raphaël

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. GOURDEAU Richard, Ph.D., président

M. COHEN Paul, Ph.D., membre et directeur de recherche

Mme PINEAU Joelle, Ph.D., membre

Mes racines, c'est Marseille...
J'y suis né, j'y ai grandi.
À Marseille, je suis chez moi.

Inspecteur Van Loc.

REMERCIEMENTS

Tout d'abord, je tiens à remercier Paul Cohen, mon directeur, pour son support et pour m'avoir offert la possibilité de travailler au sein de son laboratoire, le *Groupe de Recherche en Perception et Robotique.*

Je remercie aussi les membres du *GRPR* que j'ai pu cotoyer : Alexandre, Hai et Sousso pour leur aide précieuse ainsi que Vincent P., Jean et les autres pour leur compagnie. Une mention toute spéciale ira à Vincent Z., concepteur d'*Acropolis* et mon mentor en `C++`, sans qui ce projet n'aurait pu aboutir. Enfin, je remercie Stéphane pour son aide en programmation, mais surtout pour sa verve dans nos débats politiques enflammés.

Je remercie mes parents pour leur soutien sans faille et plein d'amour tout au long de cette aventure. Malgrè les 5994km qui nous séparent, ils ont toujours été présents quand j'ai eu besoin d'eux. Et merci à Régine et à Claire qui m'ont elles aussi soutenu à distance.

Je remercie aussi mes divers colocataires qui m'ont supporté pendant mon aventure montréalaise –Guitou, Juls, Thilde, Hélène et surtout mon partenaire de galère : Vince–. Une pensée ira à ceux aux cotés de qui j'ai fait un bout de chemin à Montréal : Laurence, Anne-Laure et Loïc, Kroute, Ruku, Marsu, Élo et tous ceux que j'ai croisés (trop) rapidement mais qui auront quand même influencé ma vie.

Mon dernier mot sera pour les amis qui sont loin, mais qui sont restés proches ; parmi eux, Ève, Seb, Pierrot (et bien-sûr Vaness), Fred, Aurèle, Alice, Jim, Boris et les autres...

RÉSUMÉ

Les applications impliquant des équipes de plates-formes robotiques mobiles autonomes sont déja nombreuses et seront amenées à se multiplier grâce aux avancées du matériel et des capacités de traitement embarquées. Contrôler de telles équipes pour atteindre des objectifs divers demande d'un coté la mise en place des outils de contrôle de chaque entité et de l'autre de coordonner ces entités de façon à atteindre le ou les but(s) commun(s). Pour le contrôle de plates-formes autonomes, l'apparition d'architectures hybrides délibératives/réactives a permis de concilier les avantages des approches délibératives, issues de l'intelligence artificielle traditionnelle, et des approches réactives ou comportementales, basées sur le concept du stimulus/réflexe.

Le travail présenté dans ce mémoire est axé sur le développement d'une architecture de coordination multi-robots intégrant les concepts de l'approche hybride. Ce système se veut configurable, flexible et capable de prendre en charge des équipes diverses. Il s'inscrit dans une stratégie de développement incrémental et modulaire initiée au *Groupe de Recherche en Perception et Robotique* de l'*École Polytechnique de Montréal.*

L'architecture se base sur un moteur de gestion de missions centralisé, basé sur un formalisme utilisant une machine à états particulière : le réseau de Pétri. Cette représentation permet de décrire, outre les objectifs et les tâches à accomplir, la façon dont les ressources doivent être utilisées. Un gestionnaire d'assignation répartit les tâches de manière optimale entre les différents membres en fonction de leurs capacités et de leur situation. Ce gestionnaire prend en compte l'évolution des équipes aux cours du temps, via l'ajout et la suppression de membres à la volée.

La base de la couche locale, qui implante un contrôle réactif commandé par le gestionnaire d'équipe, est l'architecture de prototypage rapide *Acropolis* développée

au *GRPR*. Ce cadre de développement et de test prend la forme d'un gestionnaire de modules (entrées/sorties et traitements algorithmiques atomiques) dont la configuration peut s'apparenter à la conception d'un circuit électronique numérique : les données transitent sur des bus de liaison entre les composants que sont les modules. L'adaptation de cet outil, initialement prévu pour être configuré par l'usager, est l'un des défis rencontrés au cours du projet.

Enfin, de façon à illustrer le fonctionnement d'une telle architecture, deux versions d'une mission de recherche et sauvetage (*Search-and-Rescue*) sont déployées et testées avec diverses équipes, de tailles et de compositions différentes. Ce déploiement permet de se rendre compte des étapes de la mise en place d'une mission dans le cadre de l'arhitecture développée, de la description globale de la mission sous forme d'un réseau de Pétri au développement des comportements nécessaires à l'aboutissement des tâches.

ABSTRACT

Mobile robotic teams have more and more real life applications, and the number of affected fields is going to grow with the improvements in the embedded hardware and computing capabilities. To have such teams reaching given objectives, one needs (1) to control each team member as a single robotic platform and (2) to coordinate their actions in order for them to work as a team. In autonomous mobile robotics, the hybrid deliberative/reactive approach is being used for several years. Control systems based on that concept merge the advantages of the two previous paradigms : the deliberative one, coming from traditional artificial intelligence, and the reactive or behavior-based one, using the stimulus/reflex concept.

This work focuses on the development of a coordination architecture for a mobile robotics team using an hybrid deliberative/reactive design. The architecture must enable the user to configure this system and to use it for a large panel of missions with different heterogeneous teams. This work is a part of the incremental development strategy initiated at the *Robotic and Perception Research Lab* (*GRPR*) at *École Polytechnique de Montréal.*

The core of this architecture consists of the two parts of the centralized team manager : the mission manager and the assignation manager. The mission manager runs a specialized state machine, known as Petri net. Petri nets are used as mission descriptors. This formalism enables users to describe objectives and tasks as well as the way of using available resources. The assignation manager, which is in charge of the team composition, optimally distributes the active tasks to the team members, according to their respective capabilities. This manager allows team composition changes, with on-line members additions and deletions.

The local layer implants the reactive control of every platform and is driven by the team manager. This layer is based on a rapid prototyping framework, *Acropolis*,

developed at the *GRPR. Acropolis* is designed as a plug-in manager. Each plug-in can be either I/Os or atomic algorithmic processes. Behaviors are described by configuration circuits, done by "wiring" plug-ins together like digital electronic components. In this work, changes have been made to this framework to allow the team manager to switch and configure behaviors.

Finally, an illustration of the use of the complete system is done with the set-up of a *Search and Rescue* mission. Two implantations of this well-known application are described and tested with some different teams, whith compositions varying in size and capabilities. These setups show the steps of the deployment of a multi-robot mission within the framework created in this work. The experimental and simulated tests on these missions finally validate the usability and the correctness of the whole architecture.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ALGORITHMES

LISTE DES ANNEXES

LISTE DES NOTATIONS ET DES SYMBOLES

α_x	limite inférieure d'assignation de l'entité x (modèle ou tâche)
API	*Application Programming Interface*
ATRV, *ATRV-Mini*	plates-formes robotiques roulantes utilisées au *GRPR*
β_x	limite supérieure d'assignation de l'entité x (modèle ou tâche)
Δ_x	ensemble d'options de spécialisation de l'entité x (modèle, tâche, jeton ou évènement)
DTD	*Document Type Definition*
e	désigne un évènement
IDC	*Iterative Dual Correspondance*
id_x	identificateur de l'entité x(jeton, tâche ou évènement)
j	désigne un jeton
$KM(A, B)$	application de l'algorithme de Kuhn-Munkres aux ensembles A et B
$n(t)$	nombre d'agents affectés à la tâche t
NP-difficile	sous-classe « difficile »des problèmes *Nondeterministic Polynomial-time*
m	désigne un modèle de tâche
STL	*Standard Template Library*
t	désigne une tâche
$T(m, j)$	fonction de spécialisation d'un modèle de tâche par un jeton
$type_m$	type de tâche du modèle m
XML	*eXtended Markup Language*

INTRODUCTION

La recherche en robotique mobile peut prendre de nombreuses formes, tant la diversité de ses défis est grande. De la mécanique à l'informatique, du contrôle à l'intelligence artificielle, du traitement d'images à l'interaction humain/machine, les domaines impliqués sont innombrables. Les possibilités d'applications pour la robotique mobile, grâce à l'augmentation des capacités de traitement embarquées notamment, sont de plus en plus vastes.

L'utilisation d'une équipe de plusieurs plates-formes robotiques mobiles autonomes donne encore une nouvelle dimension à ce champ de recherche. Pour de nombreuses applications, la possibilité d'utiliser plusieurs robots permet d'une part de gagner en performance, les tâches pouvant être réalisées de manière parallèle, d'autre part de réduire les difficultés techniques et, de fait, les coûts de développement. Que ce soit pour distribuer l'exploration de zones connues (surveillance, sauvetage, nettoyage...) ou inconnues (exploration spatiale, sous-marine ou en milieu hostile), pour transporter des objets lourds ou volumineux ou encore pour réaliser des tâches demandant de nombreuses capacités différentes simultanément, la formation d'équipe de robots amène d'un coté à une augmentation de l'efficacité et à une simplification de la conception des matériels, mais adresse aussi de nouveaux problèmes en terme de coopération. En effet, la notion d'équipe n'est pas évidente et unique : comme chez l'animal ou l'homo sapiens, les stratégies de fonctionnement en groupe sont multiples, de la coexistance à la structuration hierachique en passant par la coopération d'égal à égal.

Dans le monde de coopération multi-robots, de très nombreuses approches sont possibles : de la coopération passive à la coordination explicite. Les plates-formes peuvent ainsi agir indépendamment dans le même environnement en laissant émer-

ger une intelligence de groupe, ou coopérer explicitement grâce à divers mécanismes de coordination, centralisés ou distribués. Chacune de ces approches conduit à des solutions différentes dont la mise en œuvre est plus ou moins complexe, et comportant leur lot d'avantages et de limitations. Le choix d'une stratégie de coopération dépend donc pour beaucoup des objectifs et contraintes inhérents au cadre de déploiement des applications envisagées.

Positionnement

En robotique mobile, les axes de recherche sont ainsi très divers, au sein-même des laboratoires de recherche. Un des défis auxquels doivent faire face les équipes est la mise en commun des travaux de chacun de ses membres. En dépassant le cadre d'un développement incrémental, où chaque nouvelle pierre vient augmenter les fonctionnalités d'un projet de groupe, une logique de déploiement modulaire doit permettre la réutilisation quasi-immédiate des fonctionnalités dans différents projets.

Pour aider à maintenir une telle logique, le *Groupe de Recherche en Perception en Robotique* (*GRPR*) de l'*École Polytechnique de Montréal* s'est doté de plusieurs outils. Une couche d'abstraction matérielle libre nommée *Player* ([Gerkey et al., 2003]), développée par le *Robotic Research Lab* de l'*Université de Californie du Sud*, permet de séparer la gestion matérielle des différents équipements robotiques des structures de commande. Une architecture de prototypage rapide, *Acropolis* ([Zalzal, 2005]), développée au sein du laboratoire, apporte la modularité au niveau du développement logiciel du traitement et du contrôle des plates-formes robotiques. Cette architecture permet de décrire des comportements plus ou moins complexes qui peuvent animer un robot, en programmant des modules atomiques de traitement liés les un aux autres pour former des chaînes de contrôle complètes.

Pour compléter ces outils dans le cadre d'applications impliquant plusieurs plates-formes, un système de coopération doit être intégré à cet ensemble. Un tel système devrait d'une part permettre la coopération des plates-formes au sein d'une équipe et d'autre part gérer le déroulement de missions qu'on voudrait voir effectuer par les équipes ainsi formées. Le projet présenté ici propose une implantation d'un système répondant à ces objectifs, en s'intégrant aux outils de développement présents au laboratoire, notamment *Acropolis*.

Choix technologiques

Pour le contrôle de robot mobile, deux approches se sont opposées : l'approche délibérative et l'approche réactive ou comportementale. La première, qui prend ses racines dans l'intelligence artificielle classique, se base sur des systèmes décisionnels utilisant un haut niveau d'abstraction pour générer la commande, La seconde utilise pour sa part des chaînes de contrôle courtes dans lesquelles les données captées sont des stimuli activant des comportements plus ou moins réflexes. Depuis quelques années, un consensus est apparu dans le monde du contrôle robotique mobile sur des architectures hybrides réactives/délibératives, qui permettent de concilier les avantages des approches délibératives, réactives et comportementales (voir [Arkin, 1998; Murphy, 2000]). Ces architectures en couches emploient un système délibératif de haut niveau pour diriger, directement ou au travers de couches d'interface, une couche réactive ou comportementale qui contrôle la plate-forme.

L'approche comportementale existante dans *Acropolis* et la volonté de développement d'un cadre de définition de missions complexes pour des équipes de robots ont amené au choix d'un système permettant d'intégrer le concept d'architecture hybride réactive/délibérative dans un gestionnaire de coordination multi-robot. Le gestionnaire d'équipe centralisé jouera ainsi le rôle de couche délibérative et de sys-

tème de coordination entre les membres de l'équipe. Un formalisme de description et de maintien de mission devra être défini pour permettre le déploiement simple de missions multi-robots. La couche comportementale de chaque plate-forme sera basée sur *Acropolis*, sur lequel certaines adaptions seront nécessaires, pour s'intégrer à cette nouvelle architecture.

L'originalité de l'architecture présentée ici, outre son intégration à la couche comportementale *Acropolis* développée par le *GRPR*, consiste (1) à utiliser, dans la gestion et la description des missions, un formalisme utilisant des réseaux de Pétri, des machines à états offrant une plus grande flexibilité, et (2) à intégrer l'approche hybride réactive/délibérative du contrôle robotique au sein du système de coordination d'équipe.

Plan

Ce mémoire présente les différents aspects du développement de cette architecture, ainsi que le déploiement d'une application multi-robot de *recherche et sauvetage*[1] pour en illustrer le fonctionnement.

Le premier chapitre sera consacré à la définition des objectifs et au positionnement du projet. On présentera d'une part le cadre existant au *GRPR* et les objectifs à atteindre dans ce cadre et d'autre part un rapide état de l'art en contrôle de robots mobiles autonomes et en coopération multi-robot.

Le second chapitre présentera l'architecture générale et traitera des deux composantes du gestionnaire de coordination : le gestionnaire de mission et le système de gestion de l'équipe. Le fonctionnement de la gestion de mission et le formalisme as-

[1] traduction de l'anglais *Search-and-Rescue*

socié en occuperont la première partie alors que la seconde traitera des mécanismes d'assignation des tâches aux membres.

La couche comportementale construite autour de l'architecture *Acropolis* fera l'objet du troisième chapitre. On discutera notamment de l'utilisation de la couche d'abstraction matérielle *Player*, des modifications apportées à l'architecture *Acropolis* existante, et des outils développés autour de cette architecture pour l'intégrer au projet.

L'illustration du fonctionnement de l'ensemble de l'architecture sera présentée dans le quatrième chapitre, qui regroupera la description de la mission de *Search-and-Rescue* et de sa définition dans le formalisme retenu ainsi que l'ensemble des développements effectués pour en réaliser les comportements.

Enfin, le cinquième et dernier chapitre comportera une discussion sur le fonctionnement général du système et sur les résultats obtenus dans le cadre de l'application de *Search-and-Rescue.*

CHAPITRE 1

POSITIONNEMENT

Le but de ce chapitre est de placer ce projet dans son contexte. Une première partie sera consacrée à la présentation du contexte de développement au sein du laboratoire qui l'accueille, en présentant d'une part le cadre actuel et d'autre part les objectifs à atteindre. Les parties suivantes permettront de survoler rapidement les types de solutions existantes dans le domaine du contrôle de plates-formes robotiques mobiles et de systèmes multi-robots, ainsi que les concepts utilisés pour orienter le développement de l'architecture proposée.

1.1 Cadre et objectifs

1.1.1 Problématique

Pour démontrer la validité de nouveaux concepts, de nouveaux matériels, de nouveaux algorithmes ou leurs applications à certaines tâches, une équipe de recherche en robotique mobile doit non seulement développer les applications correspondant aux concepts mis en œuvre, mais aussi se doter d'une structure permettant leurs tests. Cette structure comprend toute la chaîne liant le matériel au contrôle de plus haut niveau. Dans le cadre d'applications utilisant plusieurs plates-formes robotiques simultanément pour accéder à un objectif commun, on peut identifier les différents éléments composant cette chaîne :

- *le matériel* : les plates-formes robotiques elles-mêmes, munies d'actuateurs et de capteurs divers, répondant aux besoins des applications.

- *le pilotage du matériel* : permet la communication entre les différents matériels et les structures de contrôle et de traitement.
- *le dispositif de contrôle* : gère le contrôle de chaque plate-forme en intégrant les données des capteurs et en fournissant les commandes des actuateurs.
- *la gestion d'équipe* : permet la coordination des différentes plates-formes afin d'aboutir à l'objectif commun.

Le travail du chercheur en robotique mobile peut se situer à chacun de ces niveaux, suivant les objectifs de sa recherche. Le problème est que, pour faire la démonstration de son travail, l'ensemble de ces éléments doivent être présents. Pour lui permettre de focaliser son effort sur ses objectifs personnels, il faut qu'il puisse réutiliser ce qui existe déjà : dans ce but, un cadre de développement est nécessaire pour chacun de ces éléments.

Pour un membre du *GRPR*, les trois premiers éléments de cette chaîne disposent déjà de ce cadre. L'objectif de ce projet est de proposer un cadre pour le dernier élément de la chaîne, à savoir la gestion d'une équipe de plates-formes mobiles et de valider son fonctionnement grâce à une application.

1.1.2 Cadre existant

1.1.2.1 Matériel

Les matériels en robotiques mobiles peuvent prendre de nombreuses formes. Pour permettre une évolutivité importante ainsi qu'une facilité d'intégration et de développement de nouvelles applications, les robots utilisés au *GRPR* sont formés selon la même architecture :

- une base robotique mobile

- un ensemble de capteurs et d'actuateurs additionnels
- une unité de traitement sous la forme d'un ordinateur embarqué

Les plates-formes mobiles disponibles sont d'une part des robots de type *ATRV* et *ATRV-mini*, de la société *iRobot*, et d'autre part des chaises roulantes *Quickie* distribuées par *Sunrise Medical*. Les robots embarquent d'origine des capteurs d'odométrie et des détecteurs de proximité (sonars). Quelques capteurs additionnels disponibles sont des télémètres laser, des détecteurs de proximité (sonars et infrarouges) et des dispositifs de positionnement (gyroscopes et clinomètres). Les unités de traitements sont des ordinateurs (compatibles PC, avec *Linux* comme système d'exploitation) munis d'une liaison réseau sans fil.

1.1.2.2 Pilotage du matériel : *Player*

Chaque matériel présent sur une plate-forme nécessite un pilote logiciel particulier. Embarquer tous les pilotes nécessaires dans les structures de contrôle aurait pour conséquence immédiate de devoir réécrire ces structures pour chaque changement de matériel.

Pour s'affranchir de ce problème, un logiciel libre, *Player* [Gerkey et al., 2003], développé au *USC*[1] *Robotic Research Lab*, fournit une interface entre les matériels et les structures de contrôle. Du point de vue du traitement et du contrôle, le modèle et la technologie employée dans les capteurs et les actuateurs n'ont pas d'importance. Partant de ce constat, la solution choisie dans *Player* est la définition d'un certain nombre de types de données génériques[2] et d'un cadre de développement de pilotes pour des matériels implantant des entrées et sorties de ces types. Le fonctionnement et l'utilisation faite de *Player* dans ce projet seront décrits plus précisément à la

[1] *University of South California*
[2] appelés *interfaces*

section 3.2 et en 3.3.1.4.

1.1.2.3 Contrôle d'un robot : *Acropolis*

En robotique mobile, une partie des traitements est souvent redondante d'une application à l'autre. Il arrive aussi que, pour faire des tests, on ne veuille changer qu'une petite partie de la chaîne de contrôle. Il est donc intéressant de pouvoir réutiliser de façon immédiate des parties de traitements.

Pour permettre cette réutilisation, une architecture de prototypage rapide a été conçue au *GRPR* par M. Vincent Zalzal, dans le cadre de son projet de maîtrise. *Acropolis*, dont les fondements sont décrits dans [Zalzal, 2005, Chapitre 3] et qui fait l'objet de la section 3.3 de ce document, met en place un cadre de programmation de modules d'entrée et de sortie et de modules algorithmiques atomiques. La juxtaposition de modules, dont l'exécution est encadrée par le système lui-même, liés entre eux par leurs entrées et sorties, permet de décrire des chaînes de contrôle complexes de façon intuitive. Pour le développeur d'applications robotiques, doter un robot d'un comportement consiste à écrire les modules spécifiques à son projet et à les lier aux modules génériques ou développés dans le cadre d'autres projets.

1.1.3 Objectifs

Dans le cas d'applications multi-robots, on cherche souvent à doter chaque robot de différents comportements, activables à la demande, de façon à permettre à l'équipe d'atteindre un but commun. Ce projet a pour objectif de créer un cadre de développement pour des applications multi-robots de ce type. Le cahier des charges associé au développement est le suivant :

- permettre la définition rapide de missions d'équipe diverses ;
- permettre l'intégration d'un nombre quelconque de robots aux capacités diverses au sein d'une équipe ;
- découpler la définition de mission, la composition d'équipe et la mise en oeuvre des comportements ;
- permettre la redéfinition de l'équipe par ajout/suppression de membres à la volée ;
- rendre la mission robuste à une défaillance de certains membres ;
- utiliser comme base l'architecture matérielle et logicielle décrite à la section précédente.

Pour répondre à ces objectifs, les axes de développement retenus pour ce projet et sa validation sont :

1. Le développement d'un gestionnaire d'équipe centralisé avec :
   - la définition d'un formalisme de description de missions,
   - la conception d'un système de gestion de l'état des missions,
   - la prise en charge de signaux permettant de refléter l'évolution de l'état de la mission,
   - la mise en place d'une gestion des membres de l'équipe et de leurs capacités,
   - la conception de méthodes d'assignation des comportements des robots.
2. L'adaptation de l'architecture de prototypage *Acropolis* pour la prise en charge de comportements différents nécessitant :
   - la possibilité de définition de comportements multiples pour une plate-forme,
   - la spécialisation des comportements par des paramètres,
   - la capacité du calcul d'une estimation sur les coût des comportements ainsi spécialisés,

- la gestion de commutations entre les comportements.

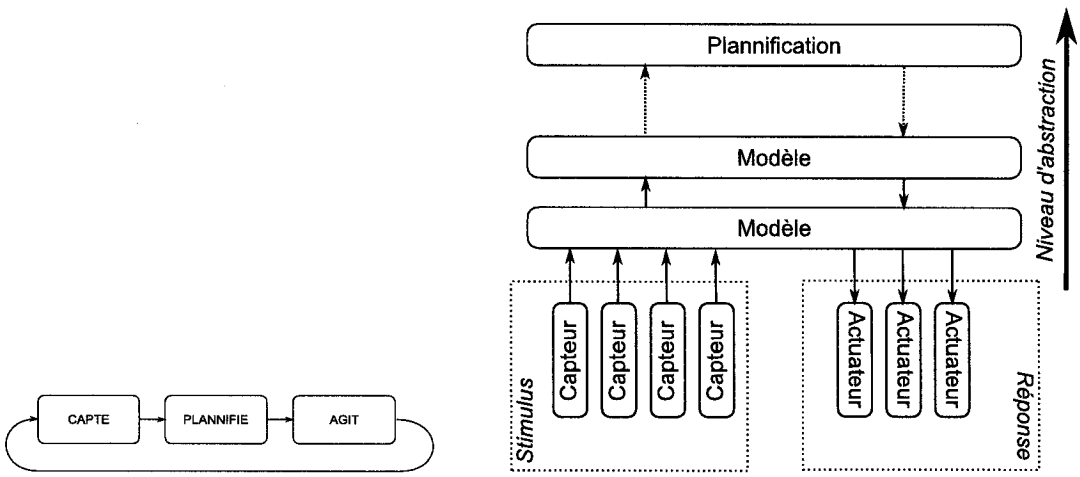
3. La création d'un module de contrôle de chaque plate-forme permettant son intégration à l'équipe et le contrôle des comportements.
4. La validation de concept grâce au développement d'une application utilisant les possibilités du système. Nous avons choisi une application de type « *recherche et sauvetage* »[3] dont le déploiement entraîne :
   - la définition de la mission sous le formalisme retenu,
   - la mise en place d'un système de cartographie partagée,
   - le développement de comportements d'exploration et de déplacement,
   - le choix de la composition d'équipes pour effectuer cette mission.

1.2 Contrôle robotique

Avant de contrôler une équipe de plates-formes robotiques dans son ensemble, il faut s'intéresser au contrôle[4] d'une plate-forme unique. Quels mécanismes mettre en jeu pour décider du déplacement ou du mouvement d'un actuateur à partir des données brutes renvoyées par les capteurs ? Dans la suite de ce chapitre, nous décrirons deux approches fondatrices dans le domaine du contrôle en robotique mobile puis le principe d'une dernière, résultant de la fusion des avantages de deux premières. Des descriptions très complètes de ces paradigmes ont été effectuées dans [Murphy, 2000] et dans [Arkin, 1998].

[3] traduction de « *search and rescue* »

[4] Le mot « contrôle » peut avoir plusieurs sens en robotique, il sera ici question du contrôle de l'entité que représente la plate-forme robotique, en passant sous silence les dispositifs de commande et d'asservissement des actionneurs.



(a) Vue temporelle (b) Vue hiérarchique

FIG. 1.1 – Contrôle délibératif : principe général

1.2.1 Contrôle délibératif

Une première approche, qui tire ses idées du domaine de l'intelligence artificielle traditionnelle, donne une place importante à la modélisation et à la représentation de la connaissance. À partir des données issues des capteurs, le système crée un modèle de son environnement, explore les possibilités, génère un plan, le traduit en actions, puis génère les commandes pour les actuateurs (figure 1.1(a)). Comme le montre la figure 1.1(b), la modélisation se fait souvent par niveaux d'abstraction croissante, des données brutes des capteurs vers une modélisation symbolique de l'environnement.

Parmi les architectures délibératives hiérarchiques ont peut citer le *Nested Hierachical Controller* (*NHC*) présenté dans [Meystel, 1993] et le *Real-time Control System* (*RCS*) développé au *NIST*[5] par Jim Albus ([Albus et Rippey, 1994]).

Une des préoccupations du concepteur de systèmes de contrôle délibératif est le

[5] *National Institute of Standards and Technology*

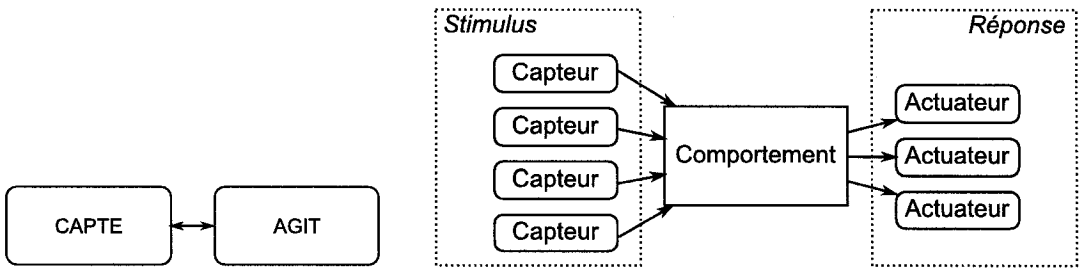
module de planification, tout en haut de la hiérarchie. La planification est un problème en soi qui ne rentre pas dans le cadre de ce projet (et dont la résolution peut être effectuée grâce à des solveurs de problème généraux[6] dont le plus connu est *STRIPS* : [Fikes et Nilsson, 1971]).

La hiérarchisation de la modélisation et de la prise de décision permet de développer des systèmes pouvant répondre à des problèmes complexes. De plus, comme dans le cas des architectures citées plus haut, ces systèmes peuvent être pensés de façon à être facilement reconfigurables. Mais deux inconvénients majeurs ont freiné l'utilisation d'architectures délibératives pures. Premièrement, la longueur du processus de contrôle, entre l'acquisition des données et la génération de la commande, en fait des systèmes peu réactifs, et donc peu adaptés aux environnement dynamiques dans lesquels les robots mobiles évoluent. Deuxièmement, la modélisation hiérarchique et la prise de décision de haut niveau rendent ces systèmes assez peu tolérants aux erreurs de modélisation.

1.2.2 Contrôle réactif

Le contrôle réactif, ou approche comportementale, est une voie importante en robotique : les limitations en capacité de calcul embarqué ont poussé à créer des chaînes de contrôle simples, des capteurs vers les actuateurs, comme le montre la figure 1.2. Un comportement génère une commande des actuateurs à partir des données des capteurs sans représentation des connaissances de l'environnement : c'est le principe du stimulus/réponse. De tels comportements sont souvent basés sur des modèles comportementaux chez les animaux : on reproduit les comportements des animaux simples (insectes, oiseaux ...). La chaîne de contrôle est souvent courte et un changement de la perception de l'environnement entraîne un changement rapide

[6] *GPS*, pour *General Problem Solver*



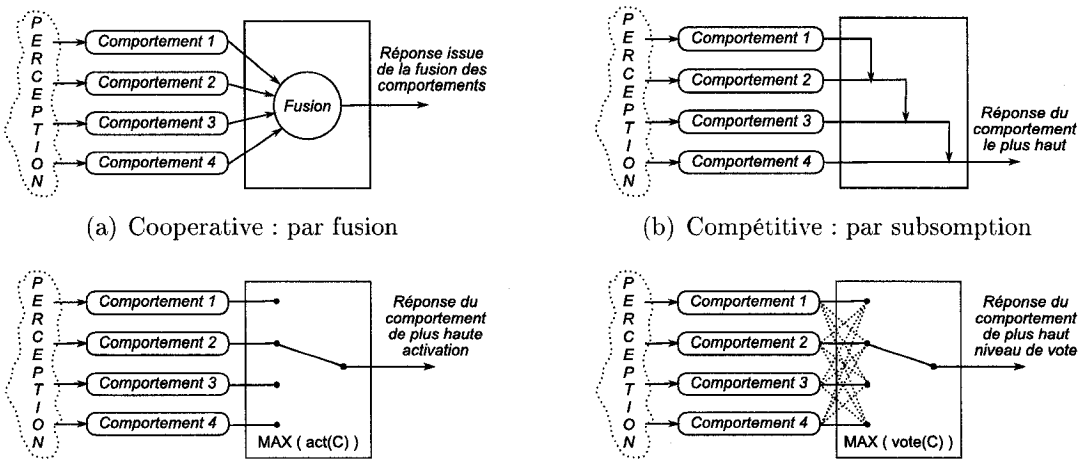
(a) Vue temporelle (b) Vue hiérarchique

FIG. 1.2 – Contrôle réactif : principe général

de l'action menée.

Un unique comportement n'est souvent pas suffisant pour donner à un robot un semblant d'autonomie. La mise en place de plusieurs comportements pose alors le problème de la coordination comportementale : chaque comportement génère une commande des actuateurs, il faut alors choisir une commande unique qui prennent en compte toutes les réponses comportementales. De nombreuses solutions sont listées dans [Pirjanian, 1999]. On peut citer deux grandes classes de coordination comportementale :

- *coopérative* : une fusion des commandes de tous les comportements est faite (figure 1.3(a)). Pour des commandes de déplacement d'une plate-forme mobile, une méthode de coordination coopérative est la superposition de champs d'attraction/répulsion ; les points d'objectifs créent un champ d'attraction vers eux tandis que les obstacles créent une répulsion ; la résultante de ces deux champs donne les directions à suivre en tout point ([Arkin, 1989]).
- *compétitive* : les comportements sont mis en compétition et la réponse du gagnant est la seule prise en compte. La sélection du comportement à suivre peut être faite par plusieurs mécanismes : par priorités statiques (exemple : l'architecture de subsomption dans [Brooks, 1986]), par degrés d'activation (exemple : [Maes, 1990]), par séquençage temporel ou encore par vote (exemple : l'architecture



(a) Cooperative : par fusion

(b) Compétitive : par subsomption

(c) Compétitive : par niveaux d'activation

(d) Compétitive : par vote

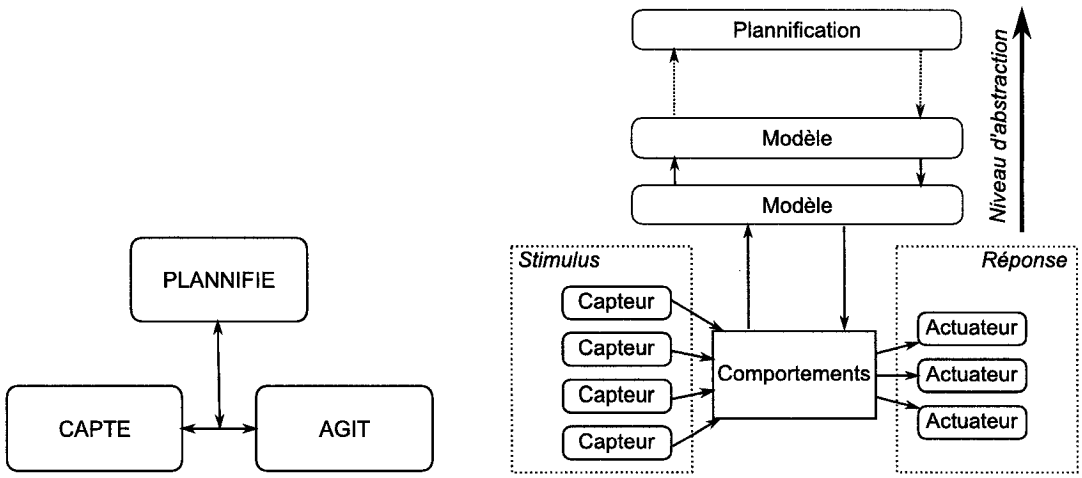
FIG. 1.3 – Contrôle réactif : coordinations comportementales

DAMN décrite dans [Rosenblatt, 1995]). Les figures 1.3(b) (c) et (d) montrent différentes coordinations comportementales compétitives.

L'approche comportementale permet de garantir un niveau élevé de réactivité. Ce type de contrôle est très efficace dans les environnements dynamiques (environnements partagés avec des humains ou avec d'autres systèmes) et dans les applications « *temps réel* ». Néanmoins le développement de tels systèmes se complexifie dès qu'on augmente le nombre de comportements possibles. Il est par ailleurs très difficile de créer des systèmes de contrôle réactif reconfigurables, ce qui tend à forcer la conception d'un système pour chaque application.

1.2.3 Contrôle hybride

Pour concilier la reconfigurabilité et la capacité à décrire des missions complexes des architectures délibératives d'une part, avec la réactivité et la tolérance aux erreurs de modèle des architectures réactives, une nouvelle approche a été développée : le contrôle hybride délibératif/réactif. Comme son nom l'indique, il s'agit d'une



(a) Vue temporelle (b) Vue hiérarchique

FIG. 1.4 – Contrôle hybride : principe général

fusion des deux approches précédentes : comme on peut voir sur la figure 1.4(b), la coordination des comportements de l'architecture réactive est contrôlée par une architecture délibérative . Il y a donc un fonctionnement parallèle du contrôle réactif et de la délibération (figure 1.4(a)).

La coordination des comportements peut être orientée par les couches délibératives de différentes façons. Dans [Arkin, 1998], ces méthodes de contrôle sont classées en quatre catégories :

- la *sélection* : le système délibératif configure les comportements actifs, choisit les comportements à activer et fixe les paramètres. La couche comportementale est reconfigurée à chaque fois que nécessaire. Un exemple de ce choix de contrôle est *AuRA*[7] dont une description assez complète est faite dans [Arkin et Balch, 1997].
- le *conseil* : la couche délibérative conseille le coordinateur comportemental, de façon à moduler la sélection des comportements. Le coordinateur utilise les principes vus en 1.2.2 et pondère son choix avec les conseils de la couche délibérative.

[7] *Autonomous Robot Architecture*

Ce type d'interaction est à la base d'*Atlantis* ([Gat, 1991]) qui utilise trois niveaux hiérarchiques indépendants, chacun conseillant le niveau inférieur : un niveau de délibération conseille un séquenceur, qui lui même conseille le contrôle réactif.

- l'*adaptation* : la couche réactive est controlée de façon continue par le système délibératif. Les réponses des comportements sont activées ou inhibées au cours de l'exécution. Pour effectuer ce contrôle continu, l'architecture *Planner-Reactor* ([Lyons et Hendriks, 1992]) utilise un planificateur « en tout temps »[8] qui fournit continûment à la couche réactive un plan dont la qualité augmente avec le temps.
- le *report de prise de décision* : dans ce type de méthodes de contrôle il n'y a pas à proprement parler de couche réactive comportementale, au sens où ce sont les plans calculés par la couche délibérative qui créent la commande finale. Le plan est gardé comme actif jusqu'à ce que de nouvelles informations ne l'invalident. Un nouveau plan est alors calculé. L'implantation la mieux connue de ce type d'architecture est *PRS*[9] ([Georgeff et Lansky, 1987]), qui suit le concept d'architecture *BDI*[10] : le système maintient des bases de croyances (issues des senseurs) et de désirs (les objectifs à accomplir) de façon à créer des intentions, qui génèrent les commandes.

1.3 Contrôle multi-robot et applications

1.3.1 Applications

Même si le nombre d'applications pour lesquelles une équipe de robots peut être employée est infini, on retrouve dans la littérature un certain nombre de type

[8] ou *anytime planner*
[9] *Procedural Reasoning System*
[10] *Beliefs-Desires-Intentions*

d'applications communes comme bancs d'essai. On peut en citer les principaux.

Le fouragement[11] et la couverture de zone Le fouragement, qui prend son nom du comportement animal de recherche de nourriture, consiste à ramener dans une ou plusieurs zones cibles tous les objets présents sur la zone de couverture. Ce type d'applications est utilisé pour effectuer des missions de nettoyage ([Santamaría et al., 1997; Parker, 1998]) ou de déminage ([Matarić et Goldberg, 2000]). La couverture de zone est une application connexe qui demande la même capacité à explorer une zone déterminée de façon à y appliquer un traitement (exemple : [Butler et al., 2000]).

Le déplacement et le transport d'objets Pour faire déplacer de gros objets (lourds ou encombrants) à des robots, deux solutions existent : dimensionner un robot unique pour lui permettre d'effectuer le déplacement seul ou utiliser une équipe de robots plus petits. La première solution impose souvent un coût de développement et de fabrication important, et peu de modularité, d'où le développement de systèmes multi-robots pour ces tâches. Deux principales classes de problèmes sont présentes : le *box-pushing* et le transport d'objets. Dans les applications de *box-pushing*, le sol est réputé plan et l'objet à déplacer peut glisser sur le sol, la coordination revient à placer les robots de façon à pousser l'objet ([Kube et al., 1993]) alors que le transport nécessite la levée et le port des objets (grâce par exemple à des bras robotiques divers comme dans [Chaimowicz et al., 2001; Huntsberger et al., 2003]).

Le déplacement en formation ou en essaim Au sein d'applications plus complexes, comme de l'exploration dirigée ou de la couverture de zone, il est souvent

[11] **traduction de l'anglais *foraging***

utile de déplacer l'ensemble des robots d'une zone à une autre. Ce déplacement peut être individuel, mais la mise en concurrence des déplacements de chacun est un frein à l'efficacité globale du déplacement. On cherche alors à reproduire des comportements animaux tels que les déplacements en formations (en ligne, ou en formation géométrique comme les oiseaux migrateurs) ou en essaims (sans configuration géométrique définie, comme certains insectes ou certains oiseaux). On peut rajouter ici la formation de *méta-robots* composés de l'agrégation dynamique de plusieurs robots simplistes, comme les *SWARM-BOTS* décrits dans [Mondada et al., 2005]).

L'exploration et la cartographie de zones Dans des zones non connues, comme par exemple lors d'une catastrophe dans un bâtiment ou une mine, ou pour la découverte de nouvelles zones, on peut chercher à répartir les robots dans l'environnement dans un but d'exploration ou de cartographie. Le plus souvent, on veut récupérer le plus d'information sur les zones accessibles : carte de l'environnement, conditions (température, toxicité ...), positionnements de points particuliers (victimes, fuites, matériaux dangereux ...). Ce domaine a aussi été très exploré, pour valider de nombreuses architectures de coordination : [Latimer et al., 2002; Rekleitis et al., 2000; Zlot et al., 2002].

Le soccer et les jeux d'équipe Un domaine plus ludique, mais considéré comme un bon banc d'essai pour les systèmes multi-robot, est l'ensemble des jeux en équipe, et plus particulièrement le *soccer*. Une compétition internationale, *RoboCup* [RoboCup, 2006], s'est imposée depuis quelques années comme une vitrine de la robotique multi-agents. De nombreux laboratoires de recherche dans l'industrie ou dans l'éducation ainsi que des amateurs participent ainsi à différents tournois selon le type de robots qui composent leur équipe de joueurs artificiels (simulation, taille réduite,

moyenne, sur quatre pattes ou humanoïdes).

La liste des types d'applications citées ici n'est bien sur pas exhaustive, et de nombreuses applications empruntent des idées de plusieurs de ces types. Dans [Farinelli et al., 2004], les auteurs donnent une classification semblable sur de nombreux travaux parus dans la littérature.

1.3.2 Niveau de coopération

Pour décrire les différences de concepts entre les différents systèmes multi-robots, il faut en étudier leur niveau de coopération. En effet, le contrôle des membres d'un système multi-robot est développé de façon à atteindre un but commun, mais l'interaction entre les membres peut se faire de multiples façons et avec des niveaux d'information partagée différents.

1.3.2.1 Niveau d'information

Une première différence qui apparaît entre les différents systèmes est la quantité d'information partagée. Certains systèmes, pour effectuer des tâches parallélisables, ont été développés en utilisant des agents ne partageant aucune information, ni même la connaissance des autres membres de l'équipe. Le développement de tels systèmes est motivé seulement par sa simplicité de conception. La plupart des systèmes multi-robots permettent un partage d'information, ou du moins donnent à chaque membre l'information d'appartenance à l'équipe.

1.3.2.2 Niveau de coordination

Dans le cas de systèmes permettant l'échange de l'information et la connaissances des équipiers, il faut se poser la question de la possibilité de coordination. La coordination est une forme de coopération dans laquelle l'action des agents prend en compte les actions des autres agent afin d'effectuer une opération globale cohérente. De nombreux systèmes multi-robots fonctionnent sans coordination : faciles à mettre en oeuvre pour des missions simples, ils permettent de répondre efficacement à des tâches où les actions ne sont pas ou peu liées entre elles.
Si la coordination est présente elle peut être plus ou moins forte : la coordination peut reposer sur un protocole de coordination explicite ou apparaître de façon implicite par l'information des équipiers sur les actions engagées.

1.3.2.3 Coordination centralisée vs. distribuée

Dans le cas d'une coordination forte –explicite–, on peut avoir deux principales approches. L'équipe peut disposer d'un chef qui prend les décisions sur les actions à effectuer par chaque membre ; il s'agit alors d'une coordination centralisée. La deuxième approche est celle d'un système de décision distribué, où chaque agent prend la décision en fonction de l'information globale de l'équipe. Les avantages de la première approche est un développement plus simple et le fait que la décision est calculée une seule fois, mais elle expose l'équipe au risque d'être stoppée en cas de défaillance du chef. Il existe de nombreuses solutions pour concilier les deux approches, introduisant une certaine redondance dans les capacités par les agents à prendre les décisions.

1.3.3 Allocation des tâches

1.3.3.1 Équipes hétérogènes vs homogènes

Pour permettre la tolérance aux fautes, c'est-à-dire à la défaillance d'un ou des membres de l'équipe, il faut que chacune des capacités nécessaires à la mission soit présente sur plus d'un robot. La redondance de ces capacités permet aussi d'accroître l'efficacité générale de l'équipe via la parallélisation des tâches. À l'extrême, dans un certain nombre d'applications, les membres sont tous identiques et présentent les mêmes capacités. Ce sont des équipes dites homogènes, chaque agent pouvant être choisi indifféremment pour chacune des tâches.

Dans le cas d'une équipe hétérogène, les membres se différencient de par leurs capacités : les mécanismes de coordination doivent donc prendre en compte ces différences pour effectuer l'allocation des tâches. L'hétérogénéité permet une plus grande diversité de capacités sans complexifier la conception des plates-formes : certains membres peuvent être spécialisés pour certaines tâches.

1.3.3.2 Optimalité de l'allocation

Afin d'allouer les tâches aux membres de l'équipe de façon optimale, il faut déterminer le sens de « l'optimalité ». Pour cela, il faut d'abord définir une mesure d'utilité. Chaque membre de l'équipe doit alors pouvoir estimer son utilité pour chaque tâche. Cette estimation peut être très simple dans le cas de tâches élémentaires, directement calculés à partir des données : par exemple, [Gerkey et Matarić, 2002] utilise la distance cartésienne pour un déplacement. À l'opposé, l'estimation de la mesure d'utilité peut être issue d'une génération de plan pour effectuer la tâche : dans [Botelho et Alami, 1999], chaque membre construit un plan suivant

ses connaissances et en estime le coût.

Pour cette mesure, deux approches équivalentes se retrouvent dans les sytèmes existants : l'utilité (à maximiser) ou le coût (à minimiser). Dans les deux cas, une mesure de l'utilité, resp. du coût, de l'ensemble de l'équipe est la somme des utilités, resp. des coûts, des couples membres/tâches. [Gerkey et Matarić, 2004] donne une formalisation de l'optimisation combinatoire associée au problème d'une telle allocation. Ce problème peut être résolu de façon approchée (sans garantie d'optimalité) par un algorithme glouton, en utilisant l'algorithme de Kuhn-Munkres (initialement présenté dans [Kuhn, 1955], étendu aux problèmes non carrés dans [Bourgeois et Lassalle, 1971]), ou encore grâce à des mécanismes d'enchères pour les systèmes distribués ([Botelho et Alami, 1999; Gerkey et Matarić, 2002]).

1.3.3.3 Types d'allocation

Les problèmes d'allocation peuvent être formulés de différentes manières : plusieurs types de problèmes d'allocation existent. D'après [Gerkey et Matarić, 2004], ils peuvent se différencier selon trois critères :

- *membres mono-tâche ou multi-tâches* : dans le second cas, chaque plate-forme peut effectuer plusieurs tâches en parallèle, alors que cela est banni dans le premier. Le problème posé par les plates-formes multi-tâches est la gestion des exclusions des tâches entre elles. On peut contourner ce problème en créant des tâches composites pour se ramener au premier cas.
- *tâches mono-robot ou multi-robots* : dans le premier cas, chaque tâche nécessite une unique plate-forme, alors que dans le second, certaines tâches peuvent nécessiter plusieurs membres. Pour résoudre le problème posé par le second cas, [Shehory et Kraus, 1995] propose de former toutes les sous-équipes comportant

le nombre de membres nécessaire aux tâches, et les considérer comme un membre unique.

- *assignation immédiate ou étendue dans le temps* : cherche-t-on à assigner les tâches en prévoyant un calendrier des tâches futures ? Le problème avec allocation étendue a été démontré NP-difficile ([Bruno et al., 1974]). [Parker, 1998] propose pour *ALLIANCE* une solution approchée, en ignorant le calendrier et en recommençant l'assignation des tâches restantes avec les membres libérés.

Axes de développement

Le cadre de développement existant avant ce projet permettait, grâce à *Acropolis* dans son état d'avancement (voir section 1.1.2.3 et 3.3), un contrôle réactif reconfigurable des plates-formes mobiles individuellement. Chaque configuration peut être alors vue comme un comportement. L'adaptation de cette architecture, de façon à gérer plusieurs comportements à la demande, permet de former la couche réactive du système de *contrôle hybride* (voir section 1.2.3) qu'on propose de développer dans ce projet.

La conception d'*Acropolis* et son adaptation au multi-comportement induisent le choix d'une coordination comportementale qui se situe entre la *sélection* (le système délibératif devra reconfigurer le comportement lorsque nécessaire) et l'*adaptation* (les comportements se différencient par l'activation/inhibition des entrées/sorties), comme décrits en 1.2.3.

Pour la gestion de l'équipe, l'un des objectifs étant de permettre la définition de missions complexes de manière simple pour des équipes hétérogènes, un *haut niveau de coordination* (voir section 1.3.2.2) est nécessaire. Pour simplifier le développement, il a été choisi d'opter pour une *coordination centralisée* (voir section

1.3.2.3), tout en se laissant la possibilité de la répartir dans un développement futur. Avec la nommencalture vue en 1.3.3.3, l'assignation des tâches repose sur le paradigme « *mono-tâche / mono-robot / assignation étendue* » . La contrainte d'unicité d'agent pour une tâche pourra être outrepassée grâce au développement pour cette tâche d'une coopération de bas niveau et à un système de contrainte sur le nombre d'agents assignés. Pour s'affranchir de l'assignation étendue dans le temps, on utilisera la simplification qui consiste à relancer juste l'assignation seulement sur les changements apparus au cours du temps.

Dans la phase actuelle de développement, la couche délibérative de l'architecture hybride est présente uniquement au niveau de l'équipe : chaque agent effectue la tâche qui lui est assignée sous en activant le comportement équivalent. Il a été prévu la possibilité d'ajouter une couche délibérative locale à chaque agent qui permettrait de définir des tâches plus complexes mettant en jeu plusieurs comportements.

Finallement, afin de mettre à l'épreuve le système développé, une application de type « recherche et sauvetage » en terrain connu a été choisie. Cette application s'approche du fouragement, tout en se démarquant par l'utilisation d'équipes hétérogènes : ceci imposera de créer des comportements de déplacements point-à-point en sus des comportements de couverture de zone, et donc de valider le système d'assignation.

CHAPITRE 2

ARCHITECTURE GÉNÉRALE ET GESTION DE L'ÉQUIPE

Ce chapitre est dédié à la présentation de l'architecture développée dans ce projet et à la description du fonctionnement de la partie délibérative.

Dans une première section, on présentera rapidement l'architecture générale du système et les liens entre les différentes composantes. Dans la suite de ce chapitre, on décrira comment une mission est représentée par un réseau de Pétri augmenté, ainsi que sa formulation sous la forme d'un arbre XML. On énoncera ensuite les principes de la gestion interne de la mission sous cette forme et la génération des tâches en cours. La dernière partie portera sur la gestion des membres de l'équipe et l'assignation des tâches.

2.1 Architecture générale

Pour répondre aux différents objectifs cités à la section 1.1.3, le système développé dans ce projet comporte une couche centralisée délibérative gérant l'équipe et une couche réactive distribuée sur l'ensemble des agents. La figure 2.1 montre le système décisionnel centralisé, représentant la couche délibérative et les différents agents de l'équipe, embarquant chacun une couche de contrôle réactif. D'après la nomenclature décrite en 1.3.2, il s'agit donc d'une coordination forte centralisée.

Le contrôle au niveau local, pour chaque agent, sera décrit en détail au chapitre 3.

La couche délibérative centralisée gère d'une part la mission à effectuer par l'équipe

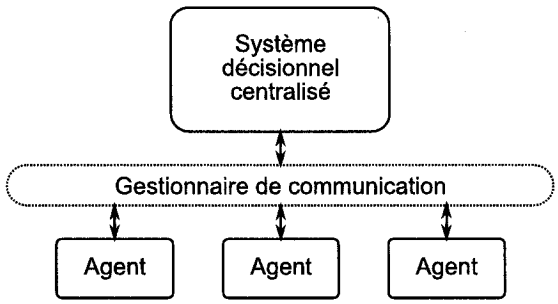


FIG. 2.1 – Vue générale de l'architecture proposée

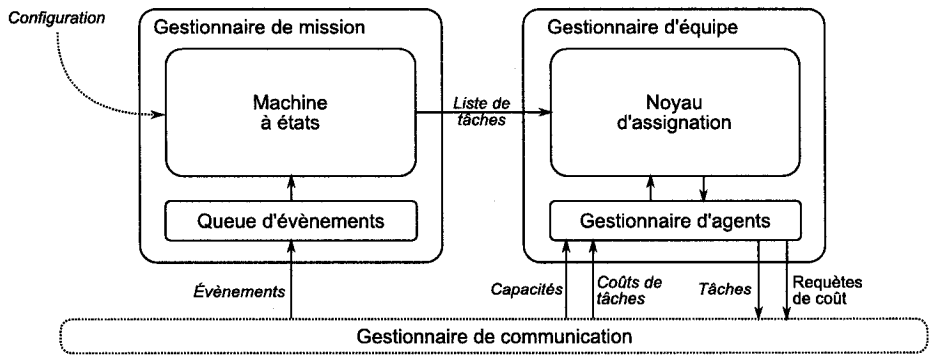


FIG. 2.2 – Composition du système décisionnel centralisé

ainsi que l'état de son avancement et d'autre part les ressources disponibles pour sa réalisation et leur assignation aux tâches nécessaires au déroulement de la mission. Pour effectuer la gestion de la mission et de son avancement, un gestionnaire de mission, constitué d'une machine à états configurable maintient l'état de la mission. Le formalisme de description et son fonctionnement seront décrits dans les deux prochaines sections. Il fournit la liste des tâches à effectuer par l'équipe immédiatement au gestionnaire d'équipe, qui se charge de leur assignation aux différents agents membres de l'équipe. Le gestionnaire d'équipe fera l'objet de la dernière section de ce chapitre. La figure 2.2 présente le découpage et les flux d'information au sein du système décisonnel.

2.2 Représentation de mission

Dans le cadre d'une architecture de gestion de mission polyvalent comme celle qu'on propose de développer ici, il est essentiel de pouvoir décrire les missions que l'on veut voir effectuer par l'équipe. Une représentation la plus générale possible est donc nécessaire pour assurer l'utilisabilité d'un tel système dans une grande diversité d'applications. La représentation choisie est fortement inspirée d'un réseau de Pétri. La description de ces réseaux de Pétri sera faite à l'aide un langage de description générique : le XML.

2.2.1 Réseau de Pétri augmenté

Un réseau de Pétri est une machine à états un peu particulière : l'état du système n'est pas donné par l'activation d'un des états prédéfinis, mais par la disposition de jetons dans un ensemble de places : le marquage. La suite de cette section s'emploiera à donner les principes généraux de fonctionnement et la spécialisation dans le cadre de ce projet ; une description complète ainsi qu'une formalisation mathématique des réseaux de Pétri est faite dans [David et Alla, 1997].

2.2.1.1 Fonctionnement général

Un réseau de Pétri est composé d'un ensemble de places et de transitions, chaque transition étant reliée à un certain nombre de places d'entrée et de places de sortie. Ces ensembles et leurs liens définissent la topologie du réseau qui détermine le comportement de la machine : la fig.2.3(a) montre un réseau de Pétri simple. L'état de la machine est donné par le marquage du réseau : il s'agit de la disposition de jetons dans les places du réseau : la fig. 2.3(b) donne un exemple de marquage.

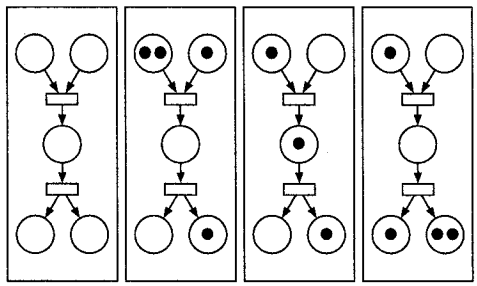


FIG. 2.3 – Exemple de réseau de Pétri : (a) Non marqué ; (b) Marqué ; (c) Après activation de la transition du haut ; (d) Après activation de la transition du bas

La modification de l'état du système se fait par l'intermédiaire d'activation de transitions dans le réseau. Une transition peut être activée si elle est activable (les principales règles d'activation seront vues en 2.2.1.4). Une transition est activable si il existe au moins un jeton dans chacune des places d'entrées. L'activation d'une transition consiste au retrait d'un jeton dans chaque place d'entrée et l'ajout d'un jeton dans chaque place de sortie : les fig. 2.3(c) et 2.3(d) montrent respectivement l'état du réseau de la fig. 2.3(b) après l'activation de la transition du haut puis du bas.

2.2.1.2 Places : modèles de tâches

A chaque place est associé un ensemble de modèles de tâches qui seront actives quand des jetons seront présents dans la place. Une tâche est une unité de fonctionnement indivisible assignée à un agent membre de l'équipe. Un modèle de tâche permet de construire les tâches et d'en définir les modalités d'assignation.

Un modèle m de tâche se compose d'un type de tâche, défini par son nom $type_m$, et d'une paire de limites –inférieure α_m et supérieure β_m– pour le nombre d'agents

associés. Le type de tâche correspond à l'action à effectuer, celle-ci pouvant être alors réalisée par plusieurs comportements différents, mais permettant d'accéder à un même but. Les limites permettront de décider du nombre d'agents à affecter à cette tâche. Un ensemble de paramètres est aussi associé à ce modèle, sous la forme d'un ensemble Δ_m de couples (nom,valeur). Ces paramètres statiques permettent de spécialiser le modèle à partir de son type.

$$m = (type_m, \alpha_m, \beta_m, \Delta_m)$$

Admettons qu'on veuille décrire, dans le cas d'une équipe de *soccer* robotisé, la tâche de défense individuelle : chaque attaquant adverse doit être « marqué » par au moins un défenseur. On veut *au moins* un agent, donc on choisira $\alpha_m = 1$; tant que des équipiers sont disponibles, on cherche à les affecter à cette tâche, donc $\beta_m = \infty$; la tâche de défense utilise un paramètre de distance de marquage *dist*. La tâche sera décrite comme :

$$def = (def\_ind, 1, \infty, \{dist = 1\})$$

2.2.1.3 Jetons : spécialisations de tâches

Chaque jeton permet la spécialisation d'un modèle de tâche en une tâche à effectuer par l'équipe.

Un jeton j est repéré par un identifiant unique id_j (un nombre unique que le système assigne à ce jeton : son ordre d'apparition) et embarque lui aussi un ensemble de paramètres Δ_j, formé de la même manière que celui des tâches.

$$j = (id_j, \Delta_j)$$

Dans le cas de notre défense individuelle, chaque jeton representera –virtuellement– un attaquant de l'équipe adverse, qui est désigné par son numéro de maillot : le 9. En admettant que ce jeton soit le 512ème créé par la machine, il sera décrit comme :

$$att = (512, \{num = 9\})$$

Pour chaque jeton et pour chaque modèle de tâche associé à la place dans lequel il se trouve, une tâche est générée avec les règles suivantes :

- son type et ses limites sur le nombre d'agents proviennent du modèle
- son identifiant est celui du jeton
- son ensemble de paramètres est construit comme l'union des ensembles du modèle et du jeton

$$t = T(m, j) = (type_m, id_j, \alpha_m, \beta_m, \Delta_t = \Delta_m \cup \Delta_j)$$

Si une place comporte le modèle de défense individuelle décrit en 2.2.1.2 et est peuplé par le jeton du numéro 9 et celui du numéro 5 : $(62, \{num = 5\})$, deux tâches seront générées :

$$(def\_ind, 512, 1, \infty, \{dist = 1, num = 9\})$$

$$(def\_ind, 62, 1, \infty, \{dist = 1, num = 5\})$$

2.2.1.4 Activations des transitions

Pour être activée, une transition doit avoir tous ses liens avec ses places d'entrées[1] activés. Il existe trois types de portes : des portes « ouvertes » en tous temps, des portes associées à des évènements et des portes temporisées.

- *portes inconditionnelles* : activées dès qu'un jeton est présent dans la place d'entrée
- *portes associées à un évènement* : activées si un évènement d'un certain type survient alors qu'un jeton est présent dans la place d'entrée
- *portes temporisées* : activées après un délai configurable suite à l'arrivée d'un jeton dans la place d'entrée

L'activation d'une transition mène à la suppression des jetons qui permettent son activation et à la création d'un nouveau jeton dans chaque place de sortie.

2.2.1.5 Spécialisations du fonctionnement

Le fonctionnement du réseau de Pétri a été spécialisé pour répondre aux besoins soulevés par son utilisation comme gestionnaire de mission.

2.2.1.5.1 Sélection de jetons par l'événement Chaque jeton correspondant à une spécialisation de tâche, il est impératif de pouvoir sélectionner celui qui doit être supprimé en fonction de l'évènement qui survient. En effet, un évènement est un message particulier qui émane d'un des agents. Chaque message comporte un

[1] les liens avec les places d'entrées seront appelés des « portes »

type, un identifiant et un ensemble de paramètres :

$$e = (type_e, id_e, \Delta_e)$$

L'identifiant correspond à l'identifiant de la tâche à laquelle l'agent est assigné. Une porte associée à un évènement est en fait associée à un type d'évènement. L'évènement peut activer la porte dont le type correspond en libérant le jeton dont l'identifiant correspond. Si aucun jeton ne correspond à cet identifiant, la porte n'est pas activée. Si l'identifiant de l'évènement ou que la porte est notée comme non sensible à l'identifiant, le premier jeton de la liste de la place d'entrée est libéré.

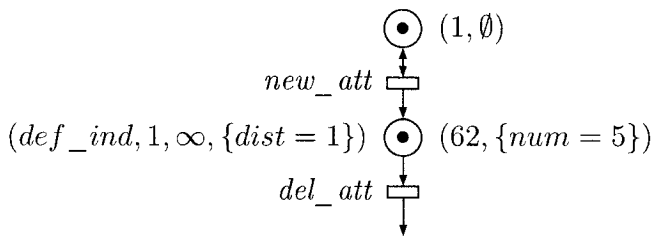
Les jetons créés par une transition dont une ou plusieurs portes est activée par évènement possèdent chacun un nouvel identifiant unique et embarquent l'ensemble des options associées aux évènements qui ont permis l'activation :

$$\Delta_j = \bigcup_{e \in vnements} \Delta_e$$

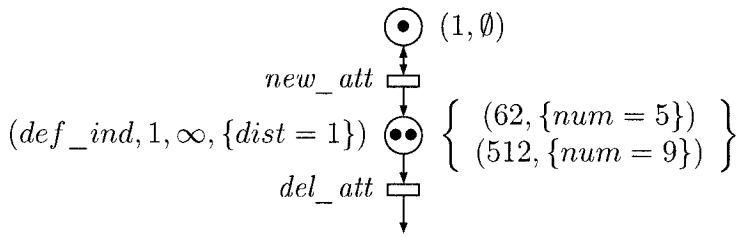
Dans le cas de notre défense individuelle, on veut qu'un jeton portant le numéro de maillot soit présent dans la place pour chaque attaquant adverse détecté. En admettant qu'on ait une place avec toujours au moins un jeton[2], une transition entre cette place et la place de défense utilisera une porte, non sensible à l'identifiant, associée à l'évènement de détection « *new_att* ». La figure 2.4(a) montre un marquage de cette partie du réseau de Pétri à un certain instant où l'attaquant numéro 5 est déja détecté.

Quand l'évènement $e = (new\_att, 7, \{num = 9\})$ survient, il active la tansition du haut. Cela permet de créer le jeton décrit en exemple du 2.2.1.3 dans la place

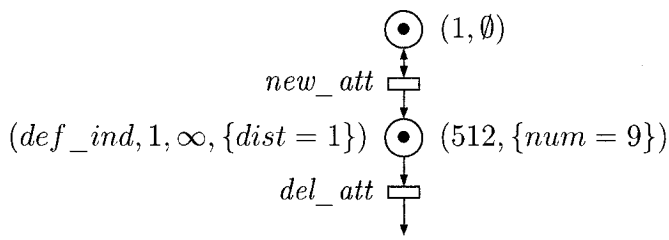
[2]on utilisera pour ça une transition bouclée, voir paragraphe suivant.



(a) État initial



(b) Après réception de l'évènement $(new\_att, 7, \{num = 9\})$



(c) Après réception de l'évènement $(del\_att, 62, \emptyset)$

FIG. 2.4 – Exemple : défense individuelle pour une application de *soccer*

centrale. La figure 2.4(b) montre le marquage après cet évènement. Si, ensuite, un des défenseurs associés à la tâche d'identifiant 62 de défense contre le joueur 5 détecte que celui ci n'est plus en position d'attaque, il emettra un évènement $e = (del\_att, 62, \emptyset)$. La transition du bas sera alors activée, le jeton est libéré et la tâche de défense contre ce joueur disparaît, comme le montre la figure 2.4(c).

2.2.1.5.2 Transitions bouclées Il arrive qu'on veuille qu'un jeton reste dans la place d'entrée même s'il permet d'activer une transition. On peut noter la place d'entrée comme place de sortie, mais le jeton serait alors un nouveau jeton (avec un nouvel identifiant), et donc générerait une nouvelle tâche dont l'assignation serait à refaire. On peut donc permettre à une transition de ne pas supprimer le jeton de certaines places d'entrée : on dira que cette transition est bouclée.

2.2.1.5.3 Actions de transitions On peut permettre au système d'envoyer des messages à certains membres de l'équipe (membres en particulier, tous, les membres à qui étaient assignées les tâches associées aux jetons supprimés, et/ou les membres à qui sont assignées les tâches associées aux jetons créés) lors d'une activation de transition. Ces messages peuvent être envoyés avant ou après la nouvelle assignation, et peuvent embarquer les paramètres associés aux jetons créés et/ou supprimés.

2.2.2 Description XML

Pour définir de tels réseaux de Pétri, on utilise un langage de description générique : le XML[3]. Ce type de langage a de multiples avantages : il permet une souplesse dans le format de description et il existe des analyseurs syntaxiques qui vérifient

[3] *eXtended Markup Language*

la validité des données et créent des structures de données faciles à utiliser par la suite.

2.2.2.1 Format de description

Le fichier de description d'un réseau comporte deux grandes parties : la description des places et de leur modèles de tâches associés, et la description des transitions, avec leur connections aux places et leurs méthodes d'activation. Une dernière section permet de déterminer le marquage initial du réseau. L'annexe II donne une description détaillée du format attendu pour décrire un réseau.

2.2.2.2 Nécessité d'un éditeur graphique

Décrire un réseau de Pétri correspondant à une mission simple est assez évident dans ce format, mais la complexité de description augmente très vite avec l'augmentation de la complexité de la mission. Un réseau de Pétri se représentant bien sous forme graphique, il est facile d'imaginer un outil d'édition graphique de tels réseaux qui générerait leurs descriptions en langage XML.

2.3 Gestion de mission

Plus que décrire les missions, les réseaux de Pétri permettent de contrôler l'avancement de la mission. En effet l'état du réseau correspond à l'état d'avancement courant de la mission : le placement des jetons présents dans le réseau détermine les tâches en cours de l'équipe et leur évolution reflète les avancements dans la mission. Il a fallu développer un système de gestion de tels réseaux décrits par l'utilisateur.

2.3.1 Représentation informatique

Comme l'ensemble du projet, ce gestionnaire est écrit en langage C++ pour fonctionner sous environnement Linux et utilise fortement les librairies de patrons standards (STL).

Au démarrage du gestionnaire, le fichier XML de configuration du réseau est interprété par la librairie `libxml++`, qui fournit un arbre XML contenant l'ensemble de la topologie du réseau. Pendant cette phase d'interprétation, la librairie vérifie que la grammaire se conforme bien au patron de configuration. Cette phase de validation utilise les définitions de type de document comme défini par la norme XML : le DTD[4], et permet d'éliminer une partie des vérifications quant à la conformité de la configuration du réseau.

L'arbre XML ainsi construit est utilisé pour construire le réseau de Pétri en mémoire. Les places sont d'abord construites et associées aux modèles de tâches. Elles seront accessibles via leur nom unique. Les transitions sont ensuite créées et liées aux places d'entrée et de sortie par des portes dont le type dépend de la configuration. La dernière étape de la construction du réseau est le marquage initial : les jetons sont positionnés dans les places initiales.

Les places et les transitions sont doublement chaînées : pour permettre un traitement rapide, chaque place liste les transitions dont elle est une entrée ou une sortie et chaque transition liste ses places d'entrée et de sortie. Cette redondance d'information permet d'une part de maintenir rapidement l'ensemble des transitions activables et d'autre part d'appliquer simplement les activations des transitions.

L'ensemble des modifications de l'état du réseau se fait de manière asynchrone sur

[4] *Document Type Definition*

une même unité d'exécution. Cette unité pilote aussi le gestionnaire d'assignation (cf. 2.4) et fonctionne sous la forme d'une boucle bloquante dont l'exécution est déclenchée par des évènements, internes ou externes. L'algorithme 2.1 montre l'action de la boucle de contrôle.

```
Toujours faire
 Attendre déclencheur
 Tant que la liste d'évènements n'est pas vide
  Effectuer les activations des transitions
 Fin Tant que
 Générer la liste des taches
 Effectuer les actions pré-assignation
 Effectuer l'assignation des tâches
 Effectuer les actions post-assignation
Fin faire
```

ALG. 2.1 – Boucle de traitement générale

Les déclencheurs de traitement pour le réseau de Pétri sont typiquement les déclencheurs d'ouverture de portes : évènements émanants des agents et signaux de temporisation. Les modifications de l'état du réseau s'effectuent tant que des transitions peuvent être activées.

2.3.2 Activation des transitions : Portes

Les portes, qui définissent les règles de passages des jetons des places d'entrées des transitions, permettent au réseau d'évoluer. Lorsque toutes les portes d'une transition sont déclenchées, la transition est activée et le réseau change d'état. Les portes actuellement implantées dans le système sont de trois types, selon leur mécanisme de déclenchement : immédiat, conditionné à la réception d'un évènement extérieur ou temporisé.

2.3.2.1 Portes inconditionnelles

Les portes les plus simples sont les portes à déclenchement immédiat, ou portes inconditionnelles : la présence d'un jeton dans la place d'entrée permet l'activation de cette porte. Une transition avec deux ou plus portes inconditionnelles permet de mettre en place le concept de « rendez-vous » : les places d'entrées représentent l'aboutissement de plusieurs sous-objectifs traités séparément, mais nécessaires à la suite de la mission. Ce type de déclenchement peut être aussi utilisé au sein d'une transition plus complexe pour vérifier une pré-condition : la place d'entrée ne doit pas être vide de jeton pour que la transition soit activée.

2.3.2.2 Portes déclenchées par évènement

Pour permettre au système de réagir face aux évènements extérieurs et à l'avancée de la mission (accomplissement des tâches, erreurs, détections quelconques...), des transitions peuvent être déclenchées sur la réception d'évènements. Ce comportement est possible grâce aux portes déclenchées par évènement : un type d'évènement est associé au lien entre place d'entrée et transition. Lorsqu'un jeton apparait dans la place, le type d'évènement est noté comme vecteur de déclenchement. Lorsqu'un évènement survient, le déclenchement peut être effectif dans un des trois cas suivants :

- un jeton portant le même identifiant que l'évènement est présent dans la place d'entrée ; dans ce cas, l'activation libèrera ce jeton.
- l'identifiant de l'évènement est l'identifiant générique ; l'activation libèrera le plus ancien jeton présent dans la place.
- la porte est notée comme non sensible aux identifiants ; l'activation libèrera là aussi le plus ancien jeton.

Lorsqu'un évènement est survenu dans ces conditions, il fait passer la porte dans un état actif. Celle-ci le restera jusqu'à ce que la transition soit activée (toutes ses portes sont actives), ou jusqu'à la disparition des jetons correspondant aux critères cités. Le gestionnaire de réseau surveille une queue d'évènements, qui est remplie par les autres unités d'exécution, et s'occupe d'effectuer les traitements sur toutes les portes qui attendent ces types d'évènement.

Les évènements permettent de spécialiser les jetons issus des transitions en leur fournissant des ensembles de paramètres –ou options– : les jetons créés dans les places de sorties embarquent l'union des ensembles d'options embarqués par les évènements qui activent la transition.

2.3.2.3 Portes temporisées

Afin de pouvoir utiliser des délais d'activation ou d'attente, des transitions peuvent être temporisées. On associe à une porte un délai d'activation. Lorsqu'un jeton apparait dans la place d'entrée, un compte à rebours est démarré. A l'échéance de ce décompte, la porte est activée, et le jeton qui a déclenché le décompte est prêt à être libéré. Une temporisation est démarrée à chaque apparition d'un jeton dans la place, donc plusieurs temporisations concurrentes peuvent être présentes au niveau d'une porte, chacune libérant son propre jeton.

2.3.3 Création de la liste des tâches

L'état du réseau donne l'avancement de la mission à l'instant présent. Cet avancement se traduit par un ensemble de tâches que l'équipe doit effectuer. Après chaque phase d'évolution du réseau, il faut donc créer à partir du marquage la liste des tâches que le gestionnaire d'équipe tentera de d'assigner aux agents membres de

l'équipe. Chaque jeton présent dans le réseau représente potentiellement une ou plusieurs tâches à effectuer par un ou plusieurs agents.

Pour construire la liste des tâches, le gestionnaire de réseau maintient la liste des jetons et les places qu'ils occupent. Pour chaque jeton, et pour chaque modèle de tâche de la place qu'il occupe, une tâche est créée dont le type est donné par le modèle, l'identifiant par le jeton, les options par l'union des options du jeton et des options du modèle et enfin les contraintes d'assignation par le modèle.

La liste ainsi créée est utilisée pour effectuer l'assignation proprement dite, de façon à respecter les limites d'assignation tout en minimisant le coût total. Cette liste permet de repérer les tâches nouvelles et celles qui disparaissent.

2.4 Gestionnaire d'assignation

Le second module du système décisionnel gère l'état immédiat de l'équipe : sa composition et l'affectation des tâches à ses membres. C'est le lien entre la vision au niveau de l'équipe donnée par le réseau de Pétri et les membres en tant qu'agents effectuant des tâches simples.

2.4.1 Gestion des membres de l'équipe

La première mission du gestionnaire d'équipe est de permettre la communication avec les agents membres de l'équipe et de maintenir l'état de sa composition. En effet, l'équipe peut évoluer au cours des missions : des membres peuvent cesser de fonctionner et de nouveaux agents peuvent se joindre à l'équipe.

Si le gestionnaire perd la connection avec un membre, à cause d'une défaillance ou

de l'arrêt de cet agent, celui-ci sera suprimé de la liste des membres, et sa tâche en cours deviendra non affectée, et donc sa réassignation deviendra possible.

De la même manière, quand un agent est capable de se connecter, c'est-à-dire s'il se trouve dans un voisinage réseau et qu'un proxy d'agent est en cours d'exécution, le gestionnaire d'équipe va demander la connection et ses capacités en terme de tâches faisables. L'agent devient alors un membre de l'équipe et est susceptible de se voir assigner des tâches de la mission. Le gestionnaire maintient donc la liste des membres A et de leurs capacités respectives. Les capacités des agents sont la liste des types de tâches qu'ils sont susceptibles d'effectuer.

Le gestionnaire d'équipe maintient aussi l'état d'affectation des agents aux tâches, et en particulier la liste des agents libres L, c'est-à-dire qui ne sont affectés à aucune tâche.

2.4.2 Requêtes de coût

Lors de leur connection, les agents fournissent au gestionnaire d'équipe leur liste de capacités, c'est-à-dire, l'ensemble des types de tâches qu'ils sont potentiellement capables d'effectuer. Or, souvent, lors de la spécialisation du modèle de tâche en tâche, les options –statiques, issues du modèle, et dynamiques, issues du jeton– donnent des informations essentielles sur cette tâche. Chaque agent capable d'effectuer ce type de tâche, de par son état, le type d'implantation de la tâche, ou encore sa position, ne demandera pas le même coût[5] pour effectuer la tâche ainsi spécialisée. Il se peut aussi que l'agent ne soit pas capable de mener à bien cette spécialisation de ce type de tâche.

[5]Le « coût » représentera ici un coût temporel, mais il est envisageable de penser à d'autres mesures de coût, comme un coût énergétique, d'usure, financier ou encore une fusion de plusieurs.

Avant chaque phase d'assignation, le système doit avoir une estimation du coût de chaque tâche pour chaque agent. Pour ce faire, pour chaque tâche, une requète de coût est envoyée à chaque agent dont les capacités contiennent le type de la tâche. La minimisation de coût sera faite avec ces estimations.

Pour éviter un blocage du système, un délai d'attente maximum est fixé pour la réception des réponses des estimations de coût. Si un agent ne répond pas pendant ce délai, il sera réputé non capable pour ces tâches et ne fera pas partie du panel d'agents pour l'assignation.

2.4.3 Assignation

La phase d'assignation correspond à l'affectation des agents issus de la liste A maintenue par le gestionnaire d'équipe aux tâches issues de la liste T générée par le réseau de Pétri. Cette liste décrit un certain nombre de tâches dont l'assignation doit respecter au mieux les contraintes sur les nombres d'agents.

2.4.3.1 Respect des contraintes

Les contraintes sur le nombre d'agents affectés à une tâche t sont définies sous la forme d'une borne inférieure α_t et d'une borne supérieure β_t. Pour être valides, les bornes doivent respecter la relation $\alpha_t \leq \beta_t$. La borne inférieure α_t peut être nulle s'il s'agit d'une tâche non prioritaire, et la borne β_t peut être infinie pour une tâche à laquelle le maximum d'agents libres doivent être affectés.

Pour suivre l'état de conformité aux contraintes, le gestionnaire d'assignation maintient une caractérisation de l'état d'assignation des tâches sous la forme de quatre ensembles, où $n(t)$ représente le nombre d'agents affectés à la tâche t :

$$\begin{aligned}
\alpha^- &= \{\, t \in T,\ n(t) < \alpha_t \} \\
\alpha^+ &= \{\, t \in T,\ n(t) > \alpha_t \} \\
\beta^- &= \{\, t \in T,\ n(t) < \beta_t \} \\
\beta^+ &= \{\, t \in T,\ n(t) > \beta_t \}
\end{aligned}$$

α^- représente l'ensemble des tâches dont le nombre d'agents qui lui sont affectés n'est pas suffisant, α^+représente l'ensemble des tâches dont l'affectation est supérieure au minimum requis, β^- liste les tâches à qui des agents peuvent être encore affectés et enfin β^+ l'ensemble des tâches dont l'affectation est supérieure au maximum. Respecter les contraintes d'affectation est équivalent à garder les ensembles α^- et β^+ vides.

Le respect des contraintes ne suffit pas à garantir le comportement attendu pour l'assignation : en effet, une assignation garantissant la relation $\forall t \in T,\ n(t) = \alpha_t$ respecte les contraintes, mais on attend de l'assignation l'affectation du plus grand nombre de tâches aux agents. Ce comportement revient d'une part à minimiser le cardinal de β^- –ce qui permet de répartir les agents à toutes les tâches– et d'autre part à maximiser $\sum n(t)$ sur β^- : celà permet de minimiser le nombre d'agents inoccupés en les affectant aux tâches non complètement assignées.

2.4.3.2 Minimisation du coût

Sous les contraintes énoncées en 2.4.3.1, on cherche à minimiser le coût total des tâches lors de l'assignation. Soit $C_{t,a}$ le coût estimé pour l'agent a pour effectuer la tâche t. Une assignation est une fonction f de l'ensemble des agents A dans

l'ensemble des tâches à effectuer T auquel est rajouté la tâche nulle[6] ω :

$$F = \{f \ : \ A \rightarrow T \cup \{\omega\}\}$$

Soit F' l'ensemble des assignations admissibles sous les contraintes énoncées en 2.4.3.1, soit le respect des bornes inférieures et supérieures pour toutes les tâches et la minimisation du nombre d'agents non assignés. F' peut alors s'exprimer :

$$F' = \left\{ f \in F, \begin{array}{c} (\forall t \in T, \alpha_t \leq Card\,(a \in A, f(a) = t) \leq \beta_t) \\ Card(a \in A, f(a) = \omega)\ minimal \end{array} \right\}$$

L'optimalité de l'assignation est donnée par la maximisation de l'utilité (discuté en 1.3.3.2). Maximiser l'utilité revient à minimiser le coût total des tâches lors de l'assignation. C'est-à-dire que l'on cherche l'assignation f_s, respectant les contraintes précédentes et dont la somme des coûts sur tous les agents soit minimale, soit telle que :

$$\forall f \in F', \sum_{a \in A} C_{f_s(a),a} \leq \sum_{a \in A} C_{f(a),a}$$

Sans la mise sous contrainte et l'affectation multiple d'agents à une même tâche, ce problème relève d'un problème d'assignation classique en optimisation combinatoire. Un algorithme efficace dans la résolution de tels problèmes est l'Algorithme Hongrois, ou algorithme de Kuhn-Munkres. On utilisera l'adaptation de cet algorithme aux problèmes dont les nombres d'agents et de tâches sont différents, décrit dans [Bourgeois et Lassalle, 1971]. Pour se ramener à ce problème et ainsi utiliser l'algorithme de Kuhn-Munkres, il faut « fixer » les tâches à assigner.

La solution proposée est d'utiliser trois phases d'assignation, chacune utilsant cet

[6]La tâche nulle ω permet de représenter la non affectation de tâche à un agent. Pour respecter la contrainte du plus grand nombre de tâches affectées, on cherchera à minimiser le nombre d'agents affectés à ω.

algorithme pour assigner un certain nombre de copies des tâches de façon à respecter les contraintes. Pour répondre à la temporalité des assignations, dans le cas de l'assignation des tâches au cours d'une mission, une assignation issue d'une précédente exécution existe déja : si cela respecte les contraintes, les tâches en cours ne doivent pas être ré-assignées.

2.4.3.2.1 Phase 1 La première phase tente de parvenir au respect de la première contrainte (le nombre d'agents assignés a chaque tâche t doit être au moins égal à sa borne inférieure α), en utilisant les agents libres (agents libres lors de la dernière assignation ou libérés par la disparition d'une tâche). C'est-à-dire qu'on applique l'algorithme de Kuhn-Munkres pour assigner les agents libres L à un nombre de copies[7] suffisant de chaque tâche pour respecter la contrainte :

$$si\ \alpha^- \neq \emptyset\ et\ L \neq \emptyset,\ KM\left(L, \bigcup_{t \in \alpha^-} \{t^{\alpha_t - n(t)}\}\right)$$

Si cette assignation ne suffit pas à respecter la première contrainte, il faut libérer des agents déja assignés en surnombre par rapport aux bornes inférieures pour tenter de respecter la contrainte. C'est la deuxième phase.

2.4.3.2.2 Phase 2 Si la première phase ne permet pas de respecter les contraintes sur les bornes inférieures, il faut libérer des agents assignés à des tâches qui dépassent la borne inférieure : les agents assignés aux tâches de α^+. Tous les agents assignés à ces tâches sont libérés et les tâches correspondantes sont à ré-assigner, avec un nombre de copies égal à leurs bornes inférieures :

$$si\ \alpha^- \neq \emptyset\ et\ \alpha^+ \neq \emptyset,\ KM\left(L \cup \{a \in A,\ f(a) \in \alpha^+\}, \bigcup_{t \in \alpha^-} \{t^{\alpha_t - n(t)}\} \cup \bigcup_{t \in \alpha^+} \{t^{\alpha_t}\}\right)$$

[7] la notation t^i sera utilisée pour représenter i copies de la tâche t

A ce stade, la contrainte due à la borne inférieure α est respectée si cela est possible avec l'ensemble des tâches à effectuer et la composition de l'équipe. Il reste alors à respecter la contrainte due au nombre maximal d'agents assignés aux tâches de β^-

2.4.3.2.3 Phase 3 La troisième phase sert à complèter si possible les assignations des tâches de β^-. On va chercher à assigner à un maximum d'agents libres des tâches de β^-. L'algorithme de Kuhn-Munkres est alors effectué entre les agents libres L et un nombre suffisant de copies de chaque tâche de β^- (pour atteindre la borne β, à concurrence du nombre d'agents libres) :

$$si\ \beta^- \neq \emptyset\ et\ L \neq \emptyset,\ KM\left(L, \bigcup_{t \in \beta^-} \{t^{min(\alpha_t - n(t), Card(L))}\}\right)$$

2.4.4 Actions Pré et Post-Assignation

Comme il a été mentionné en 2.2.1.5.3, l'activation des transitions peut être associée à des actions pré-assignation et post-assignation. Une fois que l'assignation est calculée, les actions pré-assignation sont effectuées, puis les membres sont assignés et enfin les actions post-assignation sont effectuées. Ces actions peuvent affecter un nombre quelconque de membres, décrits comme :

- une liste statique de membres nommés ;
- l'ensemble des agents de l'équipe ;
- l'ensemble des agents précédemment assignés aux tâches qui disparaissent avec les jetons disparaissant dans la transition ;
- l'ensemble des agents assignés aux nouvelles tâches correspondant aux jetons issus de la transition ;

Les actions peuvent –virtuellement– être quelconques[8]. Les actions actuellement implantée sont l'envoi de messages statiques et dynamiques. Les messages statiques utilisent le format défini pour la communication entre modules d'*Acropolis* ou entre instances. Les messages dynamiques permettent d'inclure un ensemble d'options comme l'union possible de ces ensembles :

- une liste statique d'options ;
- les options issues des jetons disparaissant dans la transition ;
- les options issues des nouveaux jetons, c'est-à-dire les options embarquées par les évènements à l'origine de l'activation de la transition.

Conclusion

Le système ainsi décrit permet de remplir les objectifs fixés en 1.1.3 : il intègre un système de définitions de missions indépendantes de la composition des équipes, permet de définir les équipes de façon dynamique en fonction des ressources disponibles et effectue un assignation des tâches optimisant le coût total.
De plus, bien que développée dans l'optique d'une interaction avec *Acropolis*, cette couche décisionnelle en est assez découplée pour être adaptable à d'autres systèmes de contrôle réactifs.

Dans le futur, plusieurs développements pourraient être envisageables pour améliorer son comportement :

- la prise en charge de missions multiples ;
- l'enregistrement de l'état de la mission, pour permettre par exemple la reprise en cas de faute ;

[8] une interface d'action est définie et permet l'ajout de nouveaux types d'actions

- la mise en place d'un système d'interaction avec l'usager, permettant le contrôle et la surveillance des missions et des membres de l'équipe.
- la prise en charge de la coexistence de plusieurs gestionnaires, avec l'inhibition ou la subordination des gestionnaires redondants, ou encore la fusion ou la mise en concurrence des missions respectives ;
- la génération automatique de missions à partir d'objectifs.

CHAPITRE 3

CONTRÔLE LOCAL : GESTION DES COMPORTEMENTS

Le gestionnaire d'équipe, délibératif et centralisé décrit au chapitre précédent, contrôle l'équipe à un haut niveau de modélisation : il décide des tâches que les agents doivent effectuer pour mener à bien la mission. Chaque agent doit alors être capable d'effectuer cette tâche. C'est le rôle de la couche réactive, qui est implantée pour chaque agent. Ce chapitre traitera de l'architecture de cette couche réactive spécialisée pour les plates-formes robotiques mobiles[1].

Dans une première partie, l'architecture générale, matérielle et logicielle, sera rapidement décrite. Une seconde partie présentera la couche d'abstraction matérielle fournie par *Player*. Suivra ensuite la description d'*Acropolis* et des modifications apportées au cours de ce projet à cette architecture de prototypage rapide afin de répondre aux besoins d'un contrôle de haut niveau des robots mobiles. Finalement seront présentés les moyens mis en place pour permettre la communication entre ces couches réactives et le gestionnaire d'équipe.

3.1 Architecture générale

Dans le cadre de ce projet, les applications utiliseront, de manière générale, des plates-formes robotiques mobiles autonomes. Ces plates-formes se composent d'un ensemble de matériels robotiques (capteurs, actuateurs ...) et d'une unité de traitement informatique sous la forme d'un ordinateur (de type compatible IBM-PC).

[1]Seul le cas de la robotique mobile sera traité ici, mais l'application à d'autres domaines est possible par remplacement des couches de plus bas niveau.

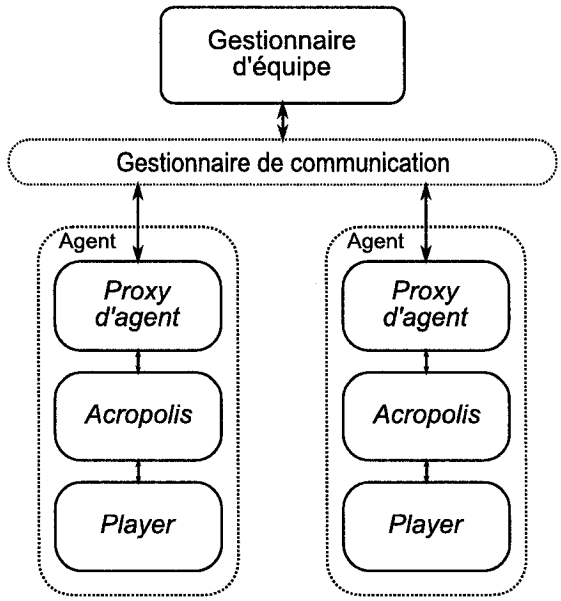


FIG. 3.1 – Architecture en couches du contrôle de la plate-forme

Pour répondre au besoin de systèmes facilement reconfigurables, permettant l'utilisation de matériels divers et capables d'effectuer un éventail de tâches extensible, un découplage entre le matériel, les divers algorithmes mis en jeu dans un contrôle réactif et le pilotage de haut niveau est nécessaire. De fait, le contrôle se fait de manière logicielle, découpé en trois couches, comme illustré pour deux agents sur la figure 3.1 :

- au niveau inférieur, la couche d'abstraction matérielle, implantée par le serveur *Player* (dont le concept et les principes de fonctionnement seront décrits à la section 3.2), qui embarque les pilotes matériels et réalise l'interface entre les couches de contrôle et les différents capteurs et actuateurs.
- au niveau intermédiaire, la couche de gestion des comportements, qui effectue les traitements et le contrôle proprement-dit de la plate-forme, réalisée par *Acropolis*. Une description de cette architecture logicielle et des modifications qu'elle a subi pendant le projet sera faite à la section 3.3.

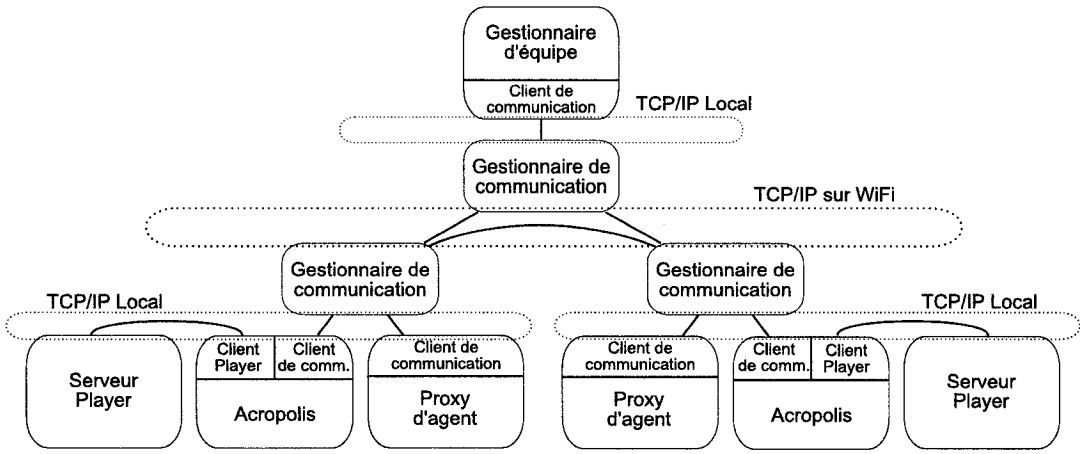


FIG. 3.2 – Fonctionnement général des communications

– au niveau supérieur, la couche de communication et de gestion de la plate-forme en tant qu'agent au service de l'équipe, décrite aux sections 3.4 et 3.5.

La communication entre les différentes couches se fait grâce au protocole TCP/IP. Typiquement, le serveur *Player*, *Acropolis* et le *proxy d'agent* sont présents sur l'unité de traitement embarquée et utilisent donc l'interface réseau locale pour communiquer. La communication entre le gestionnaire d'équipe, centralisé, et le *proxy d'agent* utilise quant à elle une connexion sans fil IEEE-802.11.
La communication entre le gestionnaire d'équipe, le *proxy d'agent* et *Acropolis* est gérée par le module de communication décrit en 3.5. La figure 3.2 montre les liens de communications entre les différents modules.

3.2 *Player* : une couche d'abstraction matérielle.

Dans un laboratoire de robotique mobile comme le *Groupe de Recherche en Perception et Robotique*, il est primordial de disposer d'outils de développement qui permettent la réutilisation des fonctionnalités d'une plate-forme robotique à une autre. De plus, la définition d'une plate-forme elle même est très fluctuante ; l'ajout

de nouveau matériel, le remplacement par des matériels équivalents ou plus performants ou encore la suppression de matériels obsolètes ne doivent pas se traduire par une perte des fonctionnalités précédemment développées. Il est donc nécessaire de découpler le matériel des différents algorithmes de contrôle et traitement.
Player [Gerkey et al., 2003] a été conçu dans le but de découpler le matériel des traitements informatiques de leurs entrées et sorties. Initialement développé au *USC*[2] *Robotics Lab*, *Player* est maintenant devenu une référence et est utilisé dans de nombreux laboratoires de robotique de par le monde. On décrira ici rapidement ses principes de fonctionnement et l'utilisation qui en est faite pour ce projet[3].

3.2.1 Principes de fonctionnement

Player est un serveur de contrôle de plates-formes robotiques. Il sert principalement de couche d'abstraction matérielle, utilisée de façon à développer les algorithmes de contrôle sans se soucier du matériel réellement utilisé. L'idée générale est la définition de types de données génériques communément présents dans les applications de robotiques, et de fournir une interface standardisée pour chaque matériel en ne prenant en compte que le type de données qu'il manipule. De cette façon, vue de l'extérieur, l'utilisation des matériels ne dépend plus de la marque, du modèle ou même de la technologie employée, mais seulement du type de capteur ou d'actuateur qu'il représente.

Player se compose d'un serveur, sous la forme d'une application embarquant les pilotes matériels, et de librairies clientes, permettant aux applications de se connecter au serveur. Le principe général de fonctionnement est illustré sur la figure 3.3.

[2] *University of South California*
[3] Pour des raison de rétro-compatibilté avec le matériel du laboratoire, deux versions majeures de *Player* (1.6.5 et 2.0.3) ont été utilisées, on se bornera ici à décrire les concepts communs aux deux versions.

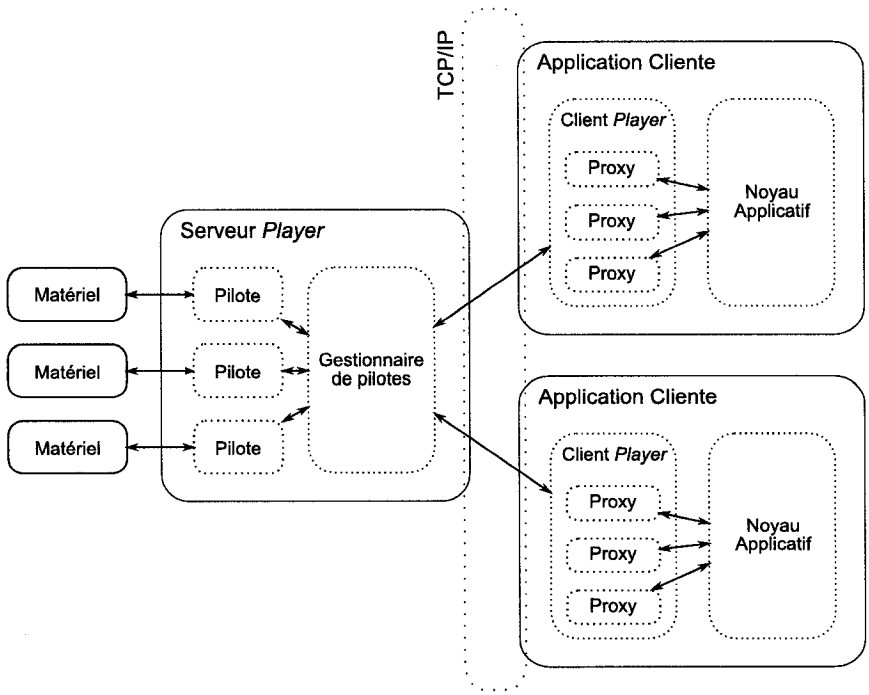


FIG. 3.3 – Fonctionnement général client/serveur *Player*

3.2.1.1 Serveur

Le serveur se compose d'un gestionnaire de pilotes, de pilotes et du serveur à proprement parler. Le gestionnaire de pilotes charge, instancie, paramétrise et contrôle les pilotes. Les pilotes sont des librairies dynamiques, répondant à une interface de programmation d'applications[4] générique. Ceux-ci implantent la communication avec le matériel et fournissent une ou plusieurs interfaces, représentant les données manipulées (il peut s'agir de données d'entrée pour les capteurs, de sortie pour les actuateurs ou les deux pour des matériels plus complexes). Pour maintenir les pilotes en constante communication avec le matériel, chacun s'exécute dans une unité d'exécution propre ; le gestionnaire de pilote a lui aussi son unité d'exécution et envoie les requêtes d'entrées et sorties sous la forme la forme de messages aux

[4] *API (Application Programming Interface)*

pilotes. L'avantage principal de cette conception en librairies dynamiques, chargées à l'exécution, est que les changements de matériels n'impliquent pas le besoin de recompilation de l'ensemble de l'application.

Le serveur met à disposition des clients toutes les interfaces que les pilotes en cours d'exécution fournissent. Ainsi, lorsqu'une application désire accéder à des données, elle se connecte via le client qu'elle embarque au serveur. Elle s'enregistre auprès de lui pour les données qui l'intéressent, en lecture (capteurs) et/ou en écriture (actuateurs). Sur la figure 3.3, on voit les flux de données, des matériels vers le serveur, en passant par les pilotes.

3.2.1.2 Clients

Pour accéder aux données, les applications doivent donc se connecter au serveur. Pour ce faire, *Player* fournit un client sous forme de librairies dynamiques et d'une interface de programmation d'application dans différents langages de programmation (C, C++, Tcl, Java, et Python). Chaque client se compose d'un module de connexion par TCP/IP au serveur, et de mandataires[5] créés pour chaque interface à laquelle l'application veut accéder. L'enregistrement d'un *proxy* peut se faire en lecture (capteur), en écriture (actuateur) ou les deux (actuateur avec retour).

Plusieurs modes de communications sont possibles entre serveur et client : l'envoi immédiat des données lors de leur disponibilité ou l'envoi sur requêtes des clients. Dans le premier cas, le serveur envoie aux clients enregistrés les données dès qu'elles sont accessibles et dans le second, ce sont les clients qui initient l'envoi des données en effectuant des requêtes au serveur.

[5]traduction de l'anglais *« proxies »*

Finalement, plusieurs clients peuvent se connecter sur le même serveur –la figure 3.3 montre l'exemple de deux clients–. Cela permet d'une part d'avoir plusieurs applications communicant avec le même matériel et d'autre part de contrôler plusieurs plates-formes dans le cadre de l'environnement de simulation *Stage* (voir 3.2.2.2).

3.2.2 Utilisation

L'utilisation de *Player* peut donc permettre de s'affranchir du contrôle de bas niveau des matériels au niveau applicatif. En plus de prendre en charge différents matériels robotiques (nativement ou par l'ajout de pilotes développés pour l'occasion), *Player* fournit un environnement de simulation via l'utilisation d'un pilote particulier, *Stage*.

3.2.2.1 Plate-forme robotique réelle

Pour fonctionner, il faut que chaque matériel utilisé ait un pilote chargé au coeur du serveur *Player*. Ce pilote doit tout d'abord exister, être instancié et finalement paramétré.
De nombreux pilotes sont distribués avec *Player*, prenant en charge des matériels robotiques en tout genre, allant des plate-formes robotiques intégrées *Pioneer* ou *iRobot* à divers modèles de caméras en passant par des télémètres laser industriels. Un patron de programmation de pilotes permet l'écriture ou l'adaptation de pilotes pour des matériels plus exotiques, ou des matériels développés sur place.
Enfin, un fichier de configuration, propre à chaque plate-forme robotique et à chaque configuration matérielle, permet de décrire les pilotes à charger et de les paramétrer (la configuration de chaque pilote lui est passé lors de son instanciation).

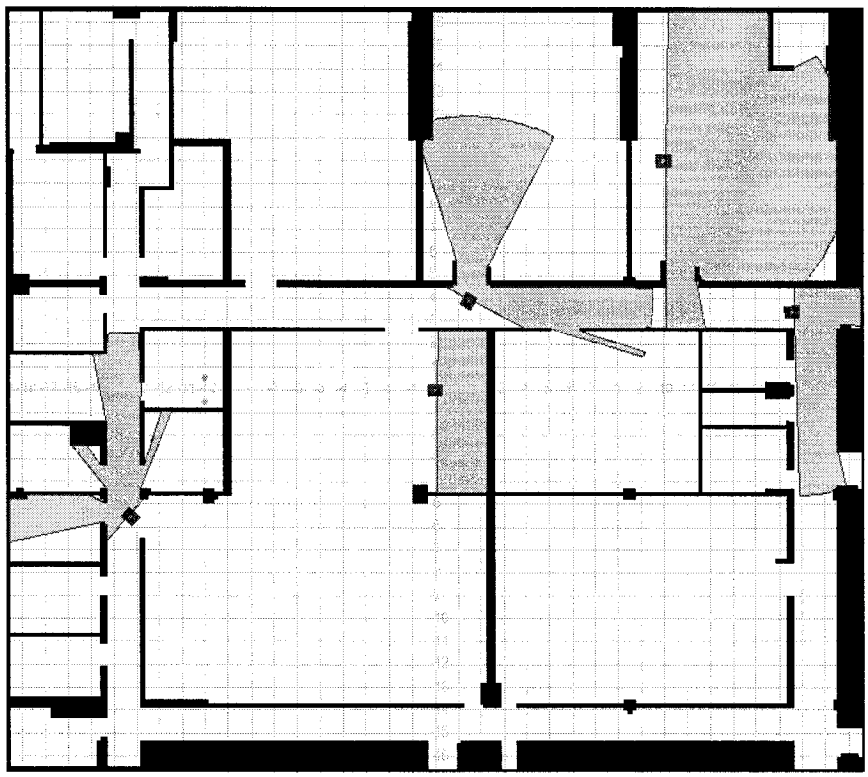


FIG. 3.4 – Exemple d'environnement de simulation dans *Stage*, avec 5 robots utilisant des télémètres laser

3.2.2.2 Environnement de simulation

Un pilote particulier, fourni avec *Player*, offre la possiblité de contrôler des plates-formes robotiques dans un environnement de simulation. *Stage*, qui se comporte du point de vue du serveur comme un pilote quelconque, permet en effet de simuler un ou plusieurs robots dans un environnement partagé, muni d'un système de visualisation (dont la figure 3.4 est une capture d'écran). Ce pilote fournit un certain nombre d'interfaces par robot, simulant des capteurs courants en robotique, comme des télémêtres laser, des sonars, des capteurs d'odométrie, simulant le contrôle de robots mobiles, de bras de préhension ou encore simulant la détection d'objets.

Comme on peut le voir sur la figure 3.4, l'environnement de simulation est composé d'une carte de murs, d'objets statiques et de plates-formes robotiques mobiles. La géométrie des robots, la géométrie et les paramètres des capteurs, ainsi que la position et la géométrie des objets sont décrits dans un fichier de description du monde de simulation. Un fichier image en noir et blanc permet de définir la carte des murs.

3.3 *Acropolis* : une architecture de prototypage rapide.

3.3.1 Description générale

Acropolis est une architecture de prototypage rapide, développée au *GRPR*. Le projet a été initié par Vincent Zalzal, dans le cadre de son projet en vue de l'obtention de sa Maîtrise ès Sciences Appliquées (voir [Zalzal, 2005]). Originellement conçu dans le cadre d'applications de robotique mobile, le projet a été généralisé au cours de ce projet pour pouvoir être utilisé hors de ce champ d'application[6].

3.3.1.1 Probèmatique

Pour répondre aux problèmes posés en 1.1.1 concernant la modularité et réutilisabilité des algorithmes au niveau du contrôle d'une plate-forme mobile, la solution la plus évidente est un découpage systématique de chaque fonctionnalité. Il s'agit d'un des paradigmes généraux en développement logiciel : le découpage en fonctions atomiques. Malgrè une application assez stricte de ce paradigme, il s'est avéré qu'une partie des fonctionnalités développées au *GRPR* n'ont pu être réutilisées

[6]grâce au découplage entre l'architecture générale et les fonctionnalités propres à ce champ d'application

et ont dû être réécrites de toutes pièces. Ce problème peut être expliqué par les difficultés persistantes pour l'utilisateur :

- le manque probable d'interopérabilité entre les différentes fonctions (formattage différent des données, utilisation de données à portée globale...)
- la nécessité de compréhension des effets sur les données des fonctions utilisées (traitements sur les entrées ou copies, gestion de la mémoire, accès concurrents aux données ...)
- le besoin de mise en place des structures de contrôle permettant l'appel de ces fonctions pour atteindre le but fixé (traitement séquentiel ou parallèle, mise en place de *rendez-vous*...)
- la difficulté d'effectuer des tests fonctionnels par partie

Acropolis a été développé pour fournir un cadre répondant à ces problèmes. L'utilisation de cette architecture et le respect de quelques paradigmes associés permettent en effet de garantir une certaine pérénité aux fonctionnalités développées au cours des projets.

Acropolis se présente sous la forme d'un gestionnaire de modules configurable. Ces modules, sous la forme de librairies dynamiques répondant à la même interface de programmation d'application (*API*), implantent chacun une fonctionalité : entrée ou sortie des données, bloc algorithmique de contrôle ou de traitement, etc ... Chacun de ces modules présente un certain nombre d'entrées et de sorties, et répond à certains messages. La configuration du gestionnaire consiste à lier les entrées et sorties des différents modules nécessaires au traitement, à la façon d'un circuit électronique ou d'un schéma *Simulink*.

Le prototypage d'une chaîne de contrôle complexe devient alors facile et rapide à mettre en place : il suffit de connecter les différents modules s'ils existent, et de

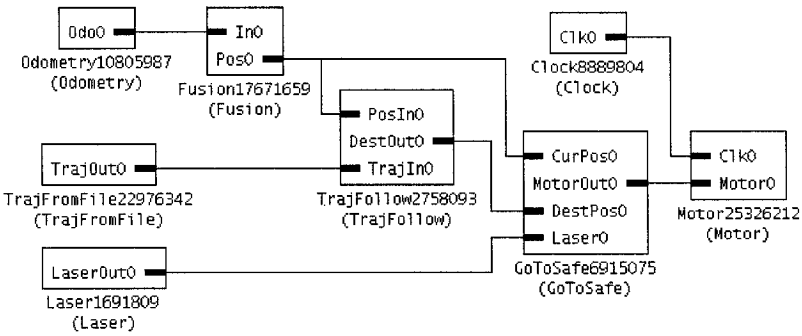


FIG. 3.5 – Exemple de circuit de configuration pour *Acropolis*

développer de nouveaux modules en gardant à l'esprit le paradigme d'atomicité. Les derniers projets développés au *GRPR* utilisent ce cadre et ces paradigmes et ont abouti à la production d'une banque de modules rétilisables.

3.3.1.2 Configuration : les circuits

La configuration du gestionnaire de modules est facilement compréhensible si on la représente sous la forme d'un circuit, à la manière d'un circuit électronique. En représentant les modules comme des blocs –l'équivalent de composants– et les liens entre entrées et sorties par des liaisons –des fils ou des pistes–, la structure de fonctionnement des chaînes de contrôle apparaît clairement. Un outil de création graphique des circuit a été développé par Alexandre Fortin. Par exemple, cet outil a permis de produire la figure 3.5, qui représente une configuration de contrôle d'une plate-forme mobile : celle-ci suit un chemin enregistré dans un fichier (*TrajFromFile*) se fiant à son odométrie (*Odometry*) et évitant localement les obstacles le long de ce chemin (*GoToSafe*) grâce a un télémètre laser (*Laser*).

Les entrées et sorties des modules sont typées : différents types génériques en rapport avec la robotique ont été définis, mais l'ajout de nouveaux types se fait très simplement. Les entrées ne peuvent être connectées qu'à des sorties de même type.

En déclarant les types de donnée de façon globale, on pousse les développeurs à les réutiliser, et donc à garantir l'interopérabilité entre les modules.

La facilité de conception de circuits permet d'imaginer et de réaliser des tests « en boîtes noires » des modules développés : on peut les intégrer à des circuits simples pour en valider le fonctionnement. Une fois validés individuellement, ils peuvent être intégrés aux chaînes de contrôle complexes.

Pour permettre aux différents modules de communiquer autrement que par l'intermédiaire d'entrées et sorties, un mécanisme de messagerie entre les modules autorise l'envoi de messages adressés. Ces messages sont surtout utiles pour de la communication sporadique (notification d'évènements, requête de configuration, etc...).

3.3.1.3 Gestion des modules

Chaque module effectue son traitement, sous la forme d'une méthode standardisée, dans son unité d'exécution propre. Les différents modules sont donc par essence en fonctionnement parallèle, mais peuvent être synchronisés grâce à leurs entrées et sorties. Le système fonctionne de manière assynchrone et la disponibilité d'une donnée sur un fil permet l'exécution du traitement associé au module dont ce fil est une entrée. Les messages entre les modules peuvent servir aussi pour la synchronisation. Ces mécanismes permettent de décharger le développeur de l'écriture des boucles de gestion des traitements et de synchronisation.

Les modules ne se partagent les données que par l'intermédiaire de leurs entrées et sorties : il y a cloisonnement des espaces de données. Cela permet de répondre aux problèmes liés à la gestion des données : les données sont locales à chaque module qui en est le seul responsable. Des mécanismes de protection ont été mis en place

pour garantir la validité des données qui transitent sur les fils :

- la mémoire tampon qui compose une sortie d'un module n'est accessible en écriture que par ce module ; lorsque le traitement est achevé par le module, l'exécution est gelée et le gestionnaire peut lire la donnée et la copier pour remplir les entrées ;
- la copie vers toutes les entrées reliées à cette sortie pouvant se faire pendant l'exécution du traitement des modules, c'est vers une mémoire tampon temporaire que la donnée est copiée ; quand le traitement finit, le tampon temporaire est noté comme tampon d'entrée et le tampon précédent devient le temporaire[7].

Ce cloisonnement des modules peut être handicapant dans certains cas : parfois, plusieurs modules devraient pouvoir partager des données ou des méthodes. Pour répondre à ce type de problème, une classe particulière de modules est gérée par le système : les *singletons*. Les singletons ont pour particularité de ne pouvoir être instanciés qu'une fois au sein d'un circuit et surtout d'être accessibles par les autres modules via un appel au gestionnaire. Le développeur de singletons doit alors prendre en charge lui-même la protection des données contre les appels concurrents.

3.3.1.4 Les modules « *Player* »

Pour utiliser *Acropolis* pour le contrôle de plates-formes robotiques pilotées par l'intermédiaire de *Player*, il a fallu développer les modules en charge de la communication avec le serveur *Player*. Pour cela, la librairie cliente C++ de *Player* a été encapsulée dans un singleton et se charge de maintenir la connexion entre le serveur et l'application. Enfin, pour chaque type de matériel, un type de donnée et un module d'entrée ou de sortie ont été développés. Chaque module crée un

[7] c'est une méthode dite de « *double buffering* »

proxy en utilisant le client (voir 3.2.1.2) et s'enregistre pour recevoir ou envoyer les données de son matériel. Sur l'exemple de la figure 3.5, on peut remarquer les modules d'entrée *Laser* et *Odometry* et le module de sortie *Motor*.

Deux ensembles de modules ont été développés au *GRPR* et sont maintenus, un pour chacune des versions de *Player* que le système devait supporter : une pour suivre le changement de version majeure *Player 2.0* et une pour la version *1.6* afin de ne pas perdre les travaux effectués avec des matériels dont la réécriture des pilotes ne fait pas partie des priorités.

3.3.2 Ajouts et spécialisations

Dans l'état de son développement au début du projet, *Acropolis* ne permettait de définir qu'un seul circuit, c'est-à-dire un seul comportement par exécution. Changer le comportement d'une plate-forme nécessitait le redémarrage du logiciel, en fournissant une nouvelle configuration. De plus, un certain nombre de fonctionnalités n'étaient que partiellement implantées. Pour répondre aux besoins du présent projet, il a fallu d'une part avancer le développement d'*Acropolis*, et d'autre part développer de nouveaux modules permettant le gestion fine des comportements :

- un module singleton concentrant les communications avec le gestionnaire d'agent ;
- un module permettant d'utiliser les paramêtres issus des tâches dans les circuits ;
- un module générique de calcul des coûts.

3.3.2.1 Gestion du multi-circuits

La prise en charge de multiples comportements au sein d'une même exécution permet, d'une part, d'accéder aux prérequis d'une gestion automatique d'une équipe

de robots comme dans le présent projet, mais aussi de penser à une commutation commandée par un utilisateur ou une autre application. Un nouveau format pour la configuration a dû être développé. Celui-ci permet de décrire plusieurs circuits fonctionnant séparément, mais partageant une partie commune. En effet, certains modules ne peuvent pas être créés plusieurs fois (les singletons par exemple, mais aussi les modules de sortie *Player*). Une base de circuit peut donc être décrite, avec les modules et les connections communs. Chaque circuit décrit ensuite utilise cette base commune et la complète en un circuit nommé implantant un type de comportement. Ce type est celui qui est utilisé dans les descriptifs de tâches à effectuer.

Du point de vue de la gestion de ces circuits :

- tous les modules sont chargés au démarrage de l'application, mais seuls ceux dont les entrées sont connectées sur le circuit courant peuvent exécuter leurs traitements ;
- le circuit fonctionnant par défaut est le circuit nommé *default* s'il existe et sinon le premier déclaré ;
- le changement de circuit peut être demandé par un module ou par message ;
- le changement implique l'invalidation des entrées et l'envoi d'un message de remise à zéro à tous les modules du nouveau circuit.

3.3.2.2 Module de gestion pour le contrôle de haut niveau

Pour éviter qu'*Acropolis* ne devienne dépendant du système de contrôle d'équipe, l'interaction entre ce système et *Acropolis* doit être extérieure au code interne de cette architecture. Pour cela, un module a été développé pour fournir cette interaction. En effet, les modifications apportées à *Acropolis* au cours de ce projet

se limitent à des améliorations et à la gestion de la commutation entre circuits. *Acropolis* peut donc continuer à fonctionner de façon autonome (sans contrôle supérieur) et cela peut permettre d'envisager un contrôle de cette commutation via une interface utilisateur ou encore par un autre système de décision.

Le module *« IAClient »*, qui est un singleton[8], permet de prendre en charge l'interaction entre le *proxy* d'agent, dont le fontionnement sera décrit à la section 3.4, et l'ensemble formé par *Acropolis* et les modules des circuits. Pour réaliser cette interaction, ce module maintient la liste des circuits et leurs modules de distribution des paramètres et de calculs de coût (cf. paragraphes suivants). Actuellement, deux types de messages provenant du *proxy* sont traités par ce module :

MSG_TYPE_TASK qui représentent les commandes de changements de circuit ; à la réception de ce type de messages, le module force le gestionnaire de modules à changer de circuit, et envoie les paramètres associés à la tâche aux modules de distribution de ce circuit ;

MSG_TYPE_TASK_COST_REQ qui représentent les requètes de coût pour une tâche ; les paramètres sont envoyés au calculateur de coût du ou des circuits qui réalisent ce type de tâche.

3.3.2.3 Module d'utilisation des paramètres

Chaque circuit réalise un type de tâche. Pour intégrer la spécialisation des tâches requises par le système au sein des circuits, il faut utiliser d'une manière ou d'une autre les paramètres qui permettent cette spécialisation. Les transmissions des données au sein des circuits se font de deux manières : via les entrées/sorties et via des messages. Le module d'utilisation des paramètres *« VarServer »* permet

[8] définition au 3.3.1.3

d'utiliser ces deux modes de communication pour distribuer les paramètres aux autres modules :

- ce module peut avoir autant de sorties que nécessaire, dont les valeurs des champs peuvent être les valeurs de paramètres nommés ou des constantes ;
- des messages peuvent être envoyés aux autres modules, dont le contenu est créé à partir d'un patron donné en configuration, complété par les valeurs des paramètres.

Chaque circuit peut avoir une ou plusieurs instances du module *VarServer*, qui recevront les paramètres lors de l'activation de ce circuit. De plus, une ou plusieurs instances de ce module peuvent être globales et recevront les paramètres à chaque changement de circuit. Lors de la réception de paramêtres, le module met à jour la valeur des paramètres reçus qu'il utilise, puis rafraîchit les sorties et envoie les messages qui dépendent des valeurs changées.

3.3.2.4 Modules de calculs de coûts

Lors de l'assignation des tâches, le système décisionnel requiert une estimation du coût de chaque tâche pour chaque agent. De plus, une tâche peut être effectuée par un même agent par des circuits différents (méthodes différentes, utilisation de matériels différent, etc...). Il faut donc que chaque circuit qui réalise le type de tâche demandé puisse en estimer le coût. C'est le rôle des modules calculateurs de coût. Un des modules du circuit doit donc réaliser ce calcul, il peut s'agir d'un module indépendant, ou d'un des modules de la chaîne de contrôle. Pour être calculateur de coût de son circuit, le module doit s'enregistrer auprès du module *IAServer*, et être capable de traiter les messages de requètes de coût. Par exemple, dans le cas d'un comportement de déplacement en terrain connu, le module de génération de

chemin devra très probablement être déclaré calculateur de coût.

Dans le cas où aucun calculateur de coût n'est enregistré auprès du module de gestion, la tâche sera réputée de coût nul. Un calculateur générique utilisant un analyseur syntaxique pour interpréter une formule appliquée aux paramètres a été envisagé, mais son manque d'utilité immédiate a poussé au report de son développement.

3.4 Proxy

Dans la même idée d'indépendance d'*Acropolis* face au système décisionnel, un programme de liaison entre le gestionnaire d'équipe et *Acropolis* a été développé. Cette couche d'interfaçage permet aussi de limiter la dépendance du système décisionnel à une couche réactive particulière. Par exemple, dans la première partie du développement du système décisionnel, un simulateur simpliste avait été mis en place pour le tester, celui-ci présentait la même interface que le proxy, mais n'utilisait pas *Acropolis*. On peut ainsi imaginer utiliser une couche de contrôle de plate-forme robotique différente avec le système décisionnel, en développant le *proxy* adéquat.

3.4.1 Lien Équipe/*Acropolis*

Le *proxy* fait donc office de mandataire pour *Acropolis* auprès du gestionnaire d'équipe. Sa mission est composée des tâches suivantes :

- enregistrer l'agent qu'il représente auprès du gestionnaire d'équipe ;
- maintenir l'ensemble des tâches que l'agent est capable d'effectuer afin d'en informer le gestionnaire ;
- transmettre les requêtes de coût à *Acropolis* et centraliser les réponses ;

- traiter les demandes d'affectation de tâches ;
- transmettre les évènements émanants de l'agent au gestionnaire de mission.

Lors de sa connexion au gestionnaire de communication (voir 3.5), le proxy essaie de communiquer avec l'instance d'*Acropolis* dont il est le mandataire, et demande la liste des circuits présents. Il en extrait alors les comportements réalisables et peut créer la liste des tâches dont il se pourra se déclarer capable (le lien entre tâche et comportement sera discuté à la section suivante). Dans le même temps, l'apparition d'une entité de type *proxy* déclenche l'envoi par le gestionnaire d'équipe d'une requête d'intégration. Si la connexion à *Acropolis* a réussi, le proxy accepte de se joindre à l'équipe et fournit la liste des tâches réalisables au gestionnaire. Ce protocole permet ainsi d'étendre l'équipe à tous les agents qui se connectent au gestionnaire de communication.

3.4.2 Affectation et requête de tâches

3.4.2.1 Tâche : circuit paramétré

Dans l'état de développement actuel, les tâches dont les agents sont capables sont directement les comportements décrits par les circuits dans *Acropolis*. Que ce soit pour estimer le coût ou pour effectivement effectuer une tâche, un seul circuit est mis en jeu, paramétré par l'ensemble d'options de spécialisation issu du gestionnaire de mission. Le rôle du *proxy* se limite donc à traduire la tâche en un comportement, c'est-à-dire en un circuit qui réalise ce comportement. Dans le cas où plusieurs circuits sont capables d'un même comportement, les requêtes de coût sont envoyées à tous, et l'affectation se fait au circuit ayant un coût minimal.

Le choix de traduction directe de tâche en comportement a été fait pour simplifier

cette couche, la définition des missions pouvant permettre de passer outre cette simplification. La présence d'un tel *proxy* dans la chaîne de propagation du contrôle ouvre néanmoins la possibilité d'insérer une couche délibérative locale plus avancée à ce niveau.

3.4.2.2 Développement futur

La simplification amenée par la traduction directe de tâche en comportement réactif au niveau local peut rendre difficile la définition de missions dans lesquelles une plate-forme doit atteindre individuellement des objectifs complexes. Le remplacement du *proxy* actuel, simple traducteur, en une couche décisionnelle locale pourrait aider dans ces cas. La définition très générique d'une tâche au niveau décisionnel global permet d'en changer la portée. Plusieurs axes de développement pourraient être explorés :

- une tâche pourrait être une séquence de comportements définie par configuration, dont les transitions se feraient par la réception de messages ;
- la couche délibérative locale pourrait présenter une machine à état, dont chaque configuration représenterait une tâche et chaque état un comportement à activer sur la couche réactive ;
- une tâches pourrait être définie sous la forme d'un ensemble d'objectifs et la couche délibérative se comporterait comme un planificateur ;
- tout autre système de délibération permettant l'activation des comportements réactifs pourrait être développé.

3.5 Communication

Dans le cadre du développement d'*Acropolis*, une couche de communication générique a été mise en place. Composée d'un réseau de gestionnaires, elle permet d'une part de s'affranchir de la gestion des interfaces de connexion[9] et d'autre part d'automatiser la détection des entités communicantes entre elles.

Un gestionnaire de communication se présente sous la forme d'un programme en tâche de fond sur chaque machine hôte d'une ou plusieurs entités communicantes (*Acropolis*, *proxy*, gestionnaire d'équipe...). Du point de vue des entités, ce programme se comporte comme un serveur, qui attend leurs connexions. Quand une entité démarre, elle s'enregistre auprès de son serveur local en fournissant un nom et un type d'entité. Le gestionnaire maintient une liste des entités qui lui sont ainsi connectées.

D'autre part, la diffusion de messages de détection entre gestionnaires permet à ceux-ci de se découvrir s'ils appartiennent à un même réseau local. A chaque nouvelle découverte, un canal de communication s'ouvre entre chaque paire, et s'opère un échange des listes d'entités. La figure 3.6 montre les connexions qui s'établissent dans le cadre d'un réseau comportant quatre hôtes possédant chacun trois entités communicantes. La diffusion restreint la possibilité de connexion aux gestionnaires se trouvant sur le même réseau local. Pour palier cette limitation, un pont logiciel a aussi été développé pour autoriser la découverte entre deux (ou plus) réseaux locaux.

Lorsque qu'une entité veut communiquer avec une autre, elle envoie son message à son gestionnaire en nommant le destinataire : soit les deux entités sont connectées au même gestionnaire (le messages transite alors sur la boucle locale via ce gestion-

[9] *sockets*

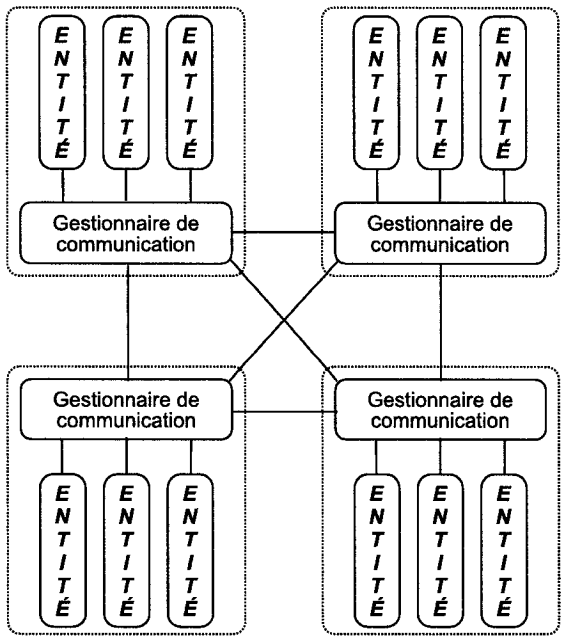


FIG. 3.6 – Exemple de schéma de connexion avec 4 hotes de 3 entités chaque

naire), soit elles sont sur des hôtes différents (le message transite du gestionnaire de l'entité source au gestionnaire de l'entité destinataire, puis à l'entité).

Le gestionnaire de communication fournit, en plus du serveur, une librairie cliente pour s'y connecter. Cette librairie permet d'associer des procédures de traitement différentes pour chaque type de message reçu. De plus, chaque client peut être notifié des connexions et des déconnexions des autres entités.

Dans le cadre d'*Acropolis*, deux types principaux de messages transitent par les gestionnaires de communication. Leur différence se situe au niveau de la portée de ces messages : les premiers sont destinés à être traités par *Acropolis*, en tant que gestionnaire de modules, alors que les seconds sont destinés aux modules eux-mêmes, désignés par leurs noms d'instance. Un autre type de message, utilisé pour la communication entre le système délibératif et les *proxies*, permet de transférer facilement les listes de paramètres. Ces messages peuvent représenter tâches,

requêtes de tâches, réponses aux requêtes, listes de capacités ou encore évènements.

Conclusion

La couche locale permet, de prendre en charge une grande diversité de plates-formes matérielles et de les doter facilement des comportements utiles aux différentes missions. Pour résumer, les objectifs suivants sont atteints :

- découplage de la prise en charge du matériel et des algorithmes de contrôles ;
- facilité de ré-utilisation et pérennité des fonctionnalités développées ;
- simplification de la définition des comportements pour les plates-formes ;
- ré-utilisibalité des comportements ainsi décrits aux diverses plates-formes ;
- possibilité de simulation des comportements avant leur application dans le monde réel.

La couche globale, quant à elle, grâce à son système de configuration et à ses méthodes d'intégration de membres hétérogènes, assure la versatilité de l'architecture au niveau du champ de définition des missions et des équipes. L'ensemble forme donc un cadre de déploiement rapide d'applications multi-robots, en fournissant une hiérarchie de contrôle complète pour une équipe de plates-formes robotiques et les outils de création des différents éléments propres à chaque projet.

CHAPITRE 4

APPLICATION : *RECHERCHE ET SAUVETAGE*

Afin d'illustrer le fonctionnement de l'architecture proposée dans le présent projet, une application de type *Search-and-Rescue*[1] a été mise en place. Ce type d'application permet de tester les différents objectifs définis dans le cadre de ce développement. En effet, pour effectuer cette catégorie de missions, plusieurs comportements différents doivent être réalisés par les membres d'une équipe qui peut être flexible et hétérogène. Le but de ce premier projet n'étant pas le déploiement des comportements d'une mission complexe, il a été choisi de se diriger vers des comportements simples, mais fonctionnels, aussi bien en simulation qu'en expérimentation.

Dans un premier temps, on décrira rapidement les objectifs et les moyens utilisés pour cette mission. On s'intéressera ensuite à la définition de la mission au niveau global dans le formalisme retenu au chapitre 2. On s'attardera finalement sur le contrôle local, qui a donné lieu au développement d'outils et de comportements pour les plates-formes.

4.1 Description

Lors de catastrophes, naturelles ou accidentelles, une des priorités des autorités est le sauvetage des personnes. Il arrive souvent que l'environnement dans lequel les secouristes doivent évoluer représente un réel danger : risque d'effondrement ou d'incendie, contamination chimique ou biologique... L'accès peut aussi être impos-

[1] le terme anglais apparaissant souvent dans la littérature francophone, on l'utilisera dorénavant pour désigner ce type d'applications

sible sans un déblaiement préalable, qui peut représenter un risque pour les victimes ensevelies. L'utilisation d'équipes de robots peut alors permettre de repérer les victimes, d'effectuer des pré-diagnostics, voire d'apporter les traitements de premier soin. La réalité de l'utilisation de robots dans ce domaine fait du *Search-and-Rescue* un axe de recherche privilégié en robotique mobile.

4.1.1 Objectifs

De nombreuses applications s'apparentent directement au *Search-and-Rescue*, du point de vue de la collaboration et de la définition globale de la mission. En effet, dans le cadre de ce projet, on ne cherche pas à démontrer l'efficacité des comportements et des matériels dans une application précise, mais à recréer les contraintes communes à ce type d'applications au point de vue de la gestion de l'équipe. Les requis de ce type d'applications sont l'exploration des zones, la découverte de points d'intérêt et une action sur ces points.

Dans le cadre du *Search-and-Rescue* classique, les points d'intérêts sont les victimes et leur détection peut être effectuée par diverses méthodes : visuelle, thermique, chimique (CO_2, phéromones...), sonore (voix, battements de coeur, respiration...), etc... Les actions à effectuer peuvent alors varier de la simple prise de données (pour permettre à une équipe de secours de préparer le sauvetage) à l'administration de premiers soins (aide respiratoire, hydratation...) en passant par le diagnostic à l'aide d'appareils de mesures.

Une autre application, qui pourrait connaître un certain essor avec l'augmentation des activités terroristes dans le monde, est la surveillance de lieux publics (aéroports, gares, halls d'exposition...) associée à la caractérisation de menaces. On peut penser alors à une équipe de robots patrouilleurs qui repéreraient des paquets sus-

pects. Les actions à mener seraient alors la caractérisation des menaces, en utilisant par exemple des robots renifleurs, et finalement leur neutralisation.

4.1.2 Contraintes et simplifications

Le matériel du laboratoire ne permettant pas d'effectuer des actions physiques sur les points d'intérêt sans demander de lourds développements, on ne cherchera pas à implanter un comportement d'action précis, mais on se bornera à dépêcher sur zone une plate-forme réputée capable de l'action.

Les locaux du laboratoire seront utilisés comme terrain d'expérimentation. Il s'agira donc d'un environnement plan et structuré. La localisation et la cartographie simultanée[2] étant un domaine de recherche à part entière, on limitera notre application à un environnement dont la carte est connue. On autorise la présence d'obstacles statiques (essentiellement les éléments de mobilier) et dynamiques (personnes, autres robots...). Cet ensemble de contraintes reflète assez bien la réalité d'une application de surveillance de lieux publics.

Les deux tâches principales requises dans ces applications sont donc l'exploration et le déplacement point-à-point, tous deux en milieu connu. L'exploration de zones planes en milieu structuré doit être associée à la détection des points d'intérêt et on se limitera donc au déplacement vers les points d'intérêt en guise d'action.

4.1.3 Environnement de simulation

Comme décrit en 3.4, *Player* fournit un environnement de simulation en deux dimensions : *Stage*. La figure 4.1 montre la carte du pavillon entourant les locaux

[2] *SLAM : Simultaneous Localization And Mapping*

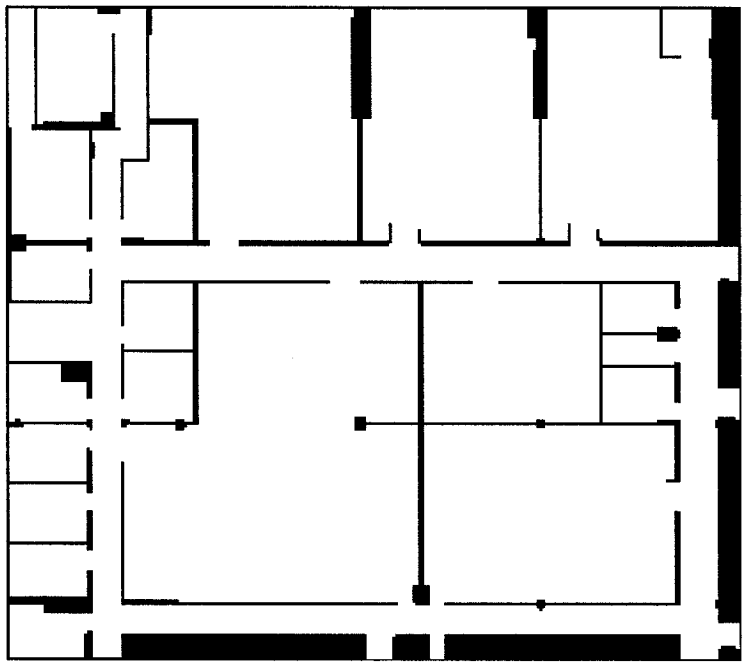


FIG. 4.1 – Carte de l'étage entourant les locaux du laboratoire

du laboratoire fournie à *Stage* pour décrire le monde dans lequel les simulations sont effectuées. De plus, un modèle de l'une de nos plates-formes matérielles, l'*ATRV-mini*, a été créé, modélisant son comportement moteur et ses principaux capteurs : un odomètre et un télémètre laser.

4.2 Définition de la mission

4.2.1 Description formelle

Une fois les objectifs définis, avec les types tâches qui seront mises en jeu dans la mission, il faut décrire la mission dans le formalisme de définition présenté en 2.2. Les deux tâches principales, exploration et déplacement sur zone, vont être associées à deux places centrales du réseau de Pétri correspondant. Au cours du déploiement

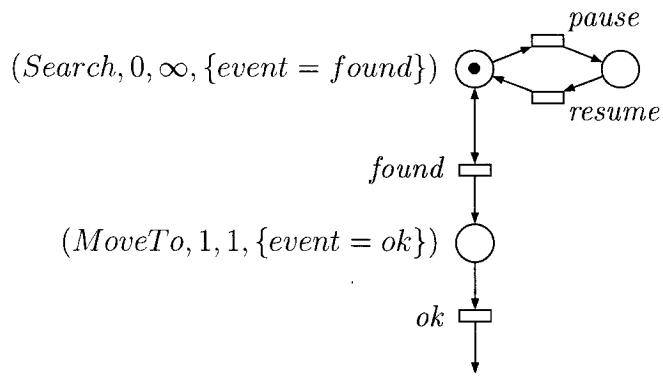
de cette application, deux approches ont été abordées quant à la manière d'effectuer l'exploration. La figure 4.2 montre les deux réseaux de Pétri décrivant les missions associées à ces deux approches.

Dans un premier temps, la tâche d'exploration a été pensée comme étant globale, c'est-à-dire qu'elle va être assignée à plusieurs plates-formes en même temps. Chaque robot utilise ses informations locales pour diriger son exploration, et l'équipe ne fournit pas d'information à chacun pour décider de la zone à explorer. La coopération entre les robots associés à cette tâche se fait de manière passive, grâce au système de cartographie distribuée (voir 4.3.1.1) La figure 4.2(a) montre le réseau de Pétri pour la mission utilisant cette exploration. Un état de pause a été rajouté, qui permet de stopper temporairement l'exploration, actionné par l'envoi par l'utilisateur d'un message *pause* et désactivé par le message *resume*.

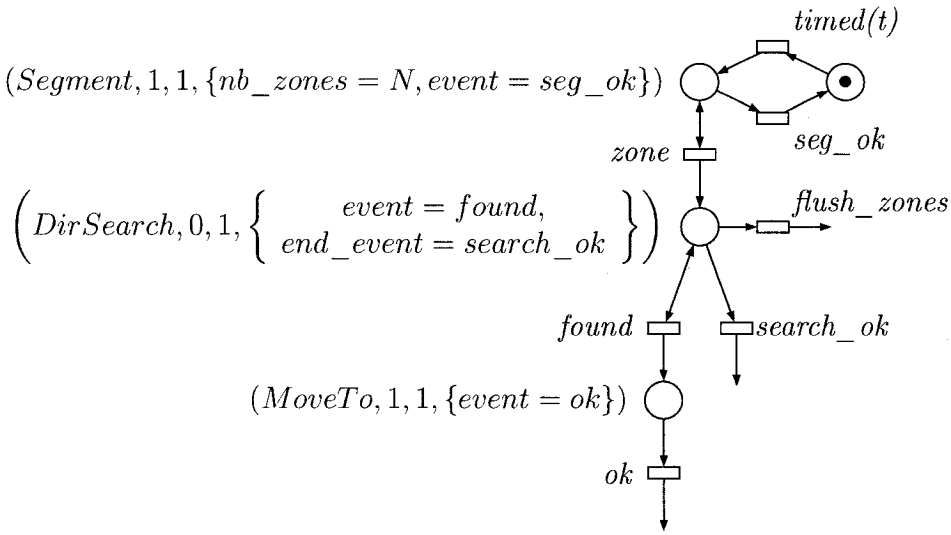
L'efficacité de cette technique d'exploration étant limitée dans les environnements structurés comme l'intérieur d'un bâtiment, une deuxième approche, donnant plus de pouvoir à l'équipe, a été explorée. À partir de la carte distribuée, un des membres de l'équipe est appelé à calculer des zones dignes d'exploration, et un seul robot explorateur est envoyé pour traiter chacune des zones. Il y a donc une nouvelle tâche, calculatoire seulement, qui apparaît dans ce cas : la segmentation en zones. La figure 4.2(b) montre le réseau de Pétri de cette version de la mission. La fonctionnalité de mise en pause a été enlevée pour faciliter la compréhension.

4.2.2 Tâches

Pour comprendre le fonctionnement de ces deux modes pour cette mission, la section suivante détaille les tâches présentes, leur paramètres, leurs contraintes d'allocation et les objectifs qu'elles doivent remplir. Leur implantation au sein de la



(a) Exploration non dirigée



(b) Exploration dirigée

FIG. 4.2 – Réseaux de Pétri associés à la mission de *Search-and-Rescue*

couche de contrôle réactif *Acropolis* sera discutée à la section 4.3.2.

4.2.2.1 Exploration

Les tâches d'exploration, bien qu'ayant les mêmes objectifs dans les deux approches proposées pour la mission, se différencient sur quelques points. L'objectif principal est la couverture de zone, accompagnée de la détection des points d'intérêt. Pour la mission décrite sur la figure 4.2(a), la répartition de la couverture de zone entre les explorateurs est passive, alors que pour celle de la figure 4.2(b), la répartition est explicite au moment de l'assignation. Dans les deux cas, la couverture de zone est associée à la détection des points d'intérêt, sanctionnée par l'envoi du message *found*, dont les options sont la position du point détecté. Ce message active la transition permettant de créer un nouveau jeton dans la place associée à la tâche d'action, sans arrêter la tâche d'exploration (transition bouclée).

Exploration non dirigée

$$(Search, 0, \infty, \{event = found\})$$

Instanciée une seule fois, la tâche d'exploration non dirigée est assignée à un maximum d'agents libres de l'équipe. Il n'y a donc pas de borne d'assignation supérieure. Il s'agit d'une tâche non prioritaire, les actions sur les points d'intérêt devant être effectuée au plus vite. La borne d'assignation inférieure est donc nulle. Pendant l'exécution de la mission, il y a toujours un et un seul jeton dans la place associée à cette tâche, sauf si la pause est demandée : dans ce cas, le jeton est « déplacé » dans la place de mise en pause.

Exploration dirigée

$$\left(DirSearch, 0, 1, \left\{ \begin{array}{c} event = found, \\ end\_event = search\_ok \end{array} \right\} \right)$$

Chaque jeton présent dans la place d'exploration dirigée représente une zone à explorer, où un seul agent peut être envoyé. La borne supérieure est donc unitaire. Là encore, la tâche n'est pas prioritaire, et la borne inférieure est nulle. Le nombre de spécialisations de la tâche dépend du nombre de zones calculées par la tâche de segmentation. Une tâche d'exploration disparaît quand l'agent finit son exploration (message *search_ok*) ou quand l'ensemble des zones est remplacé par le résultat d'une nouvelle exécution de la tâche de segmentation (message *flush_zones*).

4.2.2.2 Action sur détection

$$(MoveTo, 1, 1, \{event = ok\})$$

À chaque détection d'un point d'intérêt, un jeton est créé dans la place associée à l'action par la transition provenant de la place d'exploration. Chacune des spécialisations de cette tâche doit être assignée à un agent dès que possible. Les bornes d'assignation de cette tâche sont donc le couple $(1, 1)$. Les jetons, créés à la suite des messages de détection, ont comme options les coordonnées du point d'intérêt, ce qui permet de spécialiser la tâche. Quand la tâche sur le point d'intérêt est effectuée (avec la simplification discutée au 4.1.2, il s'agira pour la plate-forme de se placer à proximité), le jeton est libéré avec la réception du message *ok*.

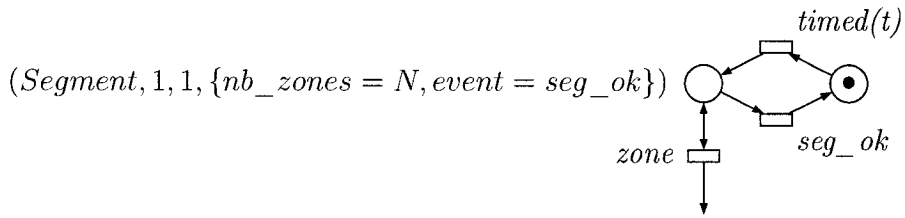
4.2.2.3 Segmentation en zones

$$(Segment, 1, 1, \{nb\_zones = N, event = seg\_ok\})$$

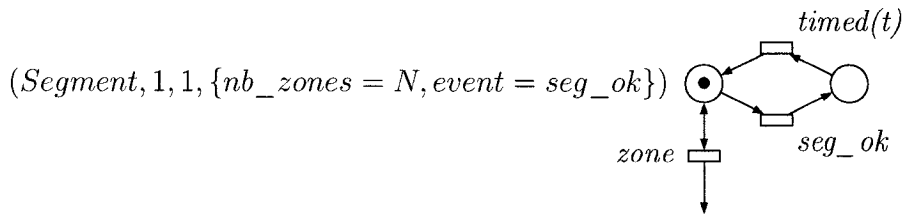
Dans le cas de l'exploration dirigée, on utilise un comportement pour découper l'ensemble des zones non explorées en segments, qui créent autant de tâches d'exploration. Un seul agent sera affecté à cette tâche. Il s'agit d'un comportement que l'on veut périodique : à une certaine fréquence, la carte doit être analysée au complet et découpée en zones. Ce comportement périodique est implanté grâce à une place sans tâche reliée à la place de segmentation par une transition temporisée (figure 4.3(a)). À chaque activation de la transition, un jeton est créé dans la place de segmentation (figure 4.3(b)). Le résultat de la segmentation est transmis grâce l'émission de *nb_zones* évènements de type *zone*, représentant chacun une segment à explorer (le résultat avec *nb_zone=2* est visible sur la figure 4.3(c)). Parallèlement, l'envoi de *nb_zones* évènements de type *flush_zones* permet de libérer les jetons correspondant aux anciennes zones à explorer. À la fin de la segmentation, le jeton est renvoyé dans la place d'attente (l'évènement *seg_ok* fait passer le réseau dans l'état de la figure 4.3(d)).

4.2.3 Évènements et messages

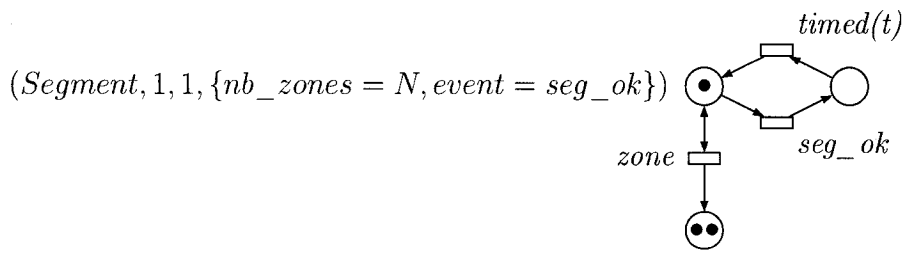
Détections Lors de l'exploration, quand un membre de l'équipe détecte un point d'intérêt, il émet un évènement *found*, dont les options situent le point, grâce à ses coordonnées. Dans une application plus avancée, on pourrait penser intégrer des informations sur le point d'intérêt, comme le type de point ou les capacités nécessaires pour traiter ce point. Pour éviter à l'équipe de détecter à nouveau le même point d'intérêt, et donc de créer une nouvelle tâche inutile, un message de



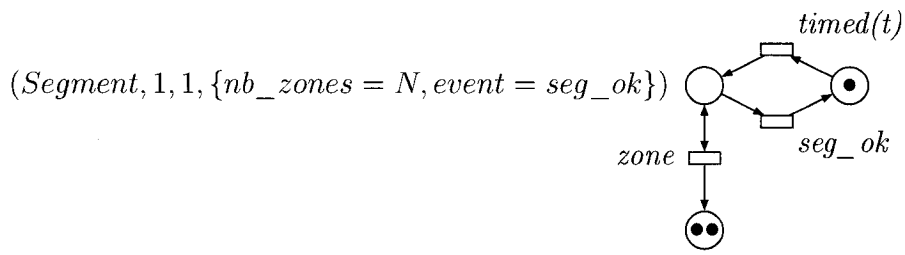
(a) État initial



(b) Après temporisation



(c) Après les messages *zone*, $N = 2$



(d) Après le message *seg_ok*

FIG. 4.3 – Comportement périodique de la segmentation

transition est envoyé à tous les membres qui ignoreront alors ce point dans leurs futures détections.

Informations de zones de recherche Dans le cas de l'exploration dirigée, l'agent en charge de la tâche de segmentation en zones de recherche doit émettre plusieurs évènements différents. Les évènements principaux sont ceux porteurs du résultat de la segmentation : pour chaque zone segmentée, un évènement de type *zone* est envoyé. Leurs options permettent de localiser chaque zone (coordonnées de son centre) et sa taille. De plus, cette segmentation doit remplacer la précédente ; il faut donc que les jetons représentant les zones précédemment segmentées soient libérés : l'émission d'un nombre équivalent d'évènements de type *flush_zones* « vide » la place d'exploration avant sa mise à jour (on notera que la transition associée est une transition non sensible à l'identificateur, voir 2.2.1.5). Une fois les zones remplacées, un dernier évènement, de type *seg_ok*, permet de signifier au gestionnaire d'équipe la fin de la segmentation.
Finalement, quand un explorateur estime l'exploration d'une zone finie, il émet un évènement *search_ok*, qui libère le jeton associé à cette zone.

4.3 Implantation des comportements

La mission étant définie, il faut que les plates-formes puissent effectuer les tâches décrites en 4.2.2. Ces tâches, réalisées par des circuits de configuration *Acropolis*, utilisent des outils communs implantés sous formes de modules. Les outils que sont la cartographie, la localisation, la planification de chemin et l'évitement d'obstacles font l'objet de la première section, alors que la description des comportements – circuits, modules particuliers et utilisation des outils– forme la seconde section.

4.3.1 Outils

4.3.1.1 Cartographie

Pour permettre d'implanter les comportements d'exploration et de déplacement en milieu connu, il faut doter les plates-formes d'une représentation de l'environnement. En clair, un système de cartographie doit être développé qui permet :

- de fournir une description géométrique plane de l'environnement ;
- de localiser chaque membre de l'équipe ;
- d'enregistrer et de partager les informations de couverture de zone ;
- de permettre la planification de chemin pour les déplacements point à point.

Pour répondre à ces objectifs, une cartographie par grille d'occupation multi-calques à été choisie. Ce système se présente sous la forme d'un module singleton d'*Acropolis*, *MapClient*, qui pourra ainsi être utilisé par les différents modules comportementaux.
Le principe de la grille d'occupation repose sur une discrétisation de l'espace plan dans lequel évoluent les plates-formes : l'espace est découpé en cases d'un damier à pas fixe, et chaque case décrit les éléments qui occupent l'espace associé. Dans notre cas, chaque case contient un ensemble d'indicateurs représentant des informations sur la case (mur, danger, zone explorée, objet...) ainsi qu'un horodatage de la dernière modification.
Pour optimiser l'espace mémoire requis par de tels tableaux, la grille est codée sous la forme de tableaux dynamiques permettant une extension bidirectionnelle sur chaque axe et utilisant un système de décalage pour se centrer sur la position effective des informations connues. La figure 4.4 montre la grille d'occupation associée à la carte des locaux du laboratoire utilisée pour les simulations (figure 4.1).

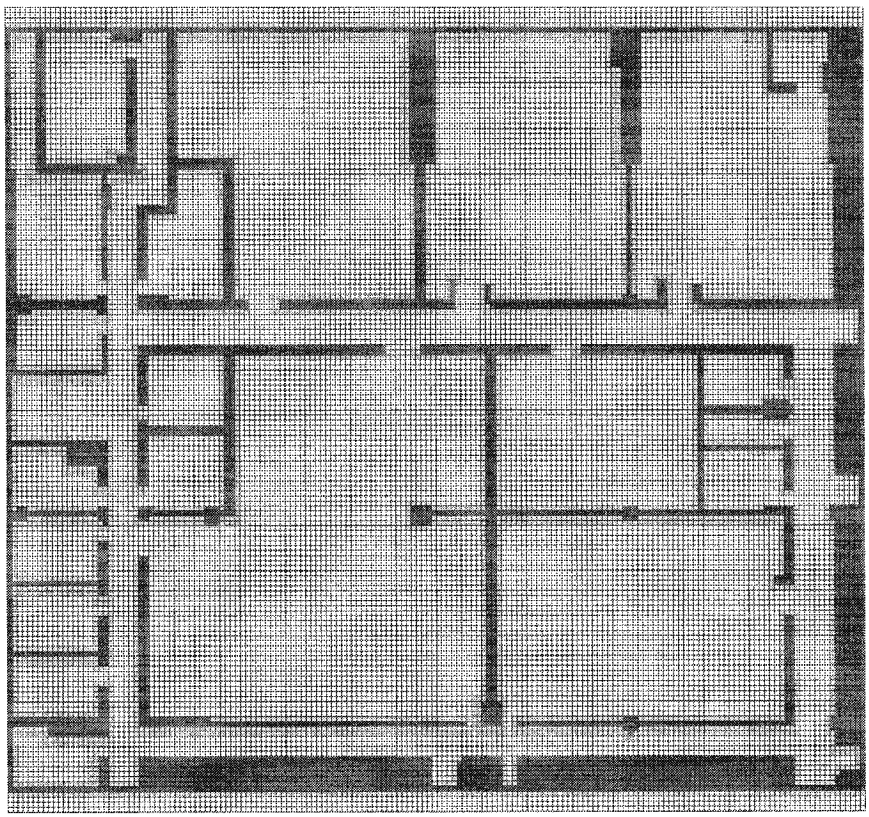


FIG. 4.4 – Grille d'occupation des locaux du laboratoires et environs

Afin d'intégrer les informations initialement connues (carte des murs) et les informations recueillies par les différents membres de l'équipe, chaque robot maintient plusieurs calques de grilles d'occupation : un calque fixe, représentant les informations données par l'utilisateur, et un calque par membre de l'équipe. Ces différents calques peuvent avoir des résolutions (taille des cases) différentes et sont reliés au calque fixe par une matrice de transformation reflétant les corrections dans la localisation des membres. Lorsqu'un agent modifie son calque, il remplit un paquet de mise à jour, envoyé périodiquement à tous les autres agents.

4.3.1.2 Localisation

Pour estimer leur position, les plates-formes robotiques mobiles peuvent utiliser des systèmes divers. Une géo-localisation utilisant des dispositifs globaux par satellite[3] permet une grande précision mais d'une part ne fonctionne pas dans les bâtiments et d'autre part génère un surcoût en matériel. La plupart des plates-formes robotiques existantes sont, par contre, équipées de capteurs d'odométrie, qui servent aussi à l'asservissement des mouvements. L'intégration des données d'odométrie permet, si l'on connaît la position initiale, d'estimer la position en tout temps. Pour affiner ces informations, on peut aussi utiliser des gyroscopes et des accéléromètres. Le problème majeur de ces dispositifs est la propagation des erreurs : au fil du temps, les erreurs de mesure s'additionnent et provoquent une localisation très erronée des plates-formes. On a estimé l'erreur relative des données d'odométrie : autour de 5% pour l'erreur linéaire et jusqu'à 10% pour l'erreur angulaire.

Plusieurs solutions ont été envisagées pour traiter ce problème :

- [Zalzal, 2005] a développé un système de relocalisation mutuelle utilisant des

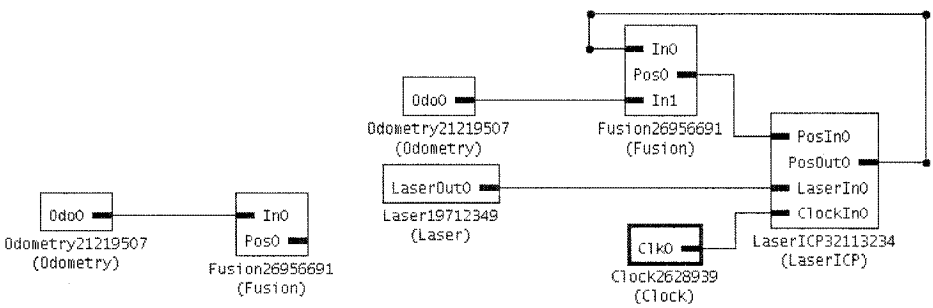
[3] *GPS* ou *GPS* différentiel

repères visuels connus détectés par des caméras omnidirectionnelles ; mais l'utilisation d'un tel système demande alors la disposition de repères visibles suffisants dans tout l'environnement, difficile à mettre en oeuvre pour l'ensemble de pièces composant l'espace d'expérimentation ;

- [Howard et al., 2003] propose d'utiliser une carte de force des signaux de communication sans fil[4] pour déterminer la position des plates-formes dans des environnements couverts par ce type d'infrastructures réseaux ; la difficulté de mise en oeuvre et la nécessité de se procurer des nouveaux matériels réseaux donnant plus d'informations ont amené à écarter cette solution ;
- [Lu et Milios, 1997] propose de recalculer la localisation en effectuant une mise en correspondance des informations données par un télémètre laser. Cette solution utilise un capteur déjà présent sur les plates-formes, demande un effort limité d'implantation au sein de l'architecture de prototypage et donne de bons résultats.

Dans [Lu et Milios, 1997], l'environnement est inconnu et la correction de la localisation se fait par mise en correspondance des nouveaux points avec les points des précédentes acquisitions. L'algorithme a donc été modifié de façon à utiliser la carte connue pour la mise en correspondance. L'algorithme, appelé *ICP* pour *Iterative Closest Point*, tente de minimiser itérativement la distance des points détectés aux points connus en appliquant une correction à la position de la plate-forme. Cette correction se fait en deux phases par itération : chaque point de l'acquisition est apparié au point connu le plus proche, puis une correction sous la forme d'une transformation géométrique (translation et rotation) est calculée grâce à la méthode des moindres carrés. Dans notre cas, la mise en correspondance se fait entre chaque point de l'acquisition et la case représentant un obstacle (mur, objet ...) la plus proche. L'algorithme s'arrête quand la correction calculée est inférieure à la

[4] *WiFi, norme ISO 802.11*



(a) Sans correction (b) Avec correction par *IDC*

FIG. 4.5 – Portion de circuit dédiée à la localisation

moitié du pas de grille de la carte ou au bout d'un nombre maximal d'itérations. [Lu et Milios, 1997] montre que cette méthode donne de bons résultats pour la correction en translation, mais converge plus lentement pour la correction de l'orientation[5]. Pour palier ce problème, une deuxième méthode d'appariement est proposée : l'*IMRP* pour *Iterative Matching Range Point.* La mise en correspondance ne se fait plus avec le plus proche voisin, mais avec le voisin qui serait à la distance la plus semblable du centre d'acquisition. Avec cette mise en correspondance, la correction en orientation est très efficace, mais la translation devient instable. La solution proposée alors, l'*IDC* pour *Iterative Dual Correspondance*, est de combiner les deux méthodes, en appliquant à chaque itération la correction en translation donnée par l'*ICP* et la correction en orientation donnée par l'*IMRP.*

Pour corriger la position des plates-formes, une boucle de rétro-action a été insérée dans le circuit de positionnement dans la partie commune des comportements implantés dans *Acropolis.* La correction est appelée périodiquement (toutes les secondes), et la position corrigée remplace la position courante (le module *Fusion* intègre les données d'odométrie et accepte les corrections issues d'autres modules). La figure 4.5 montre la partie du circuit correspondante, sans correction (a) et avec

[5]or l'odométrie génère une erreur importante en orientation

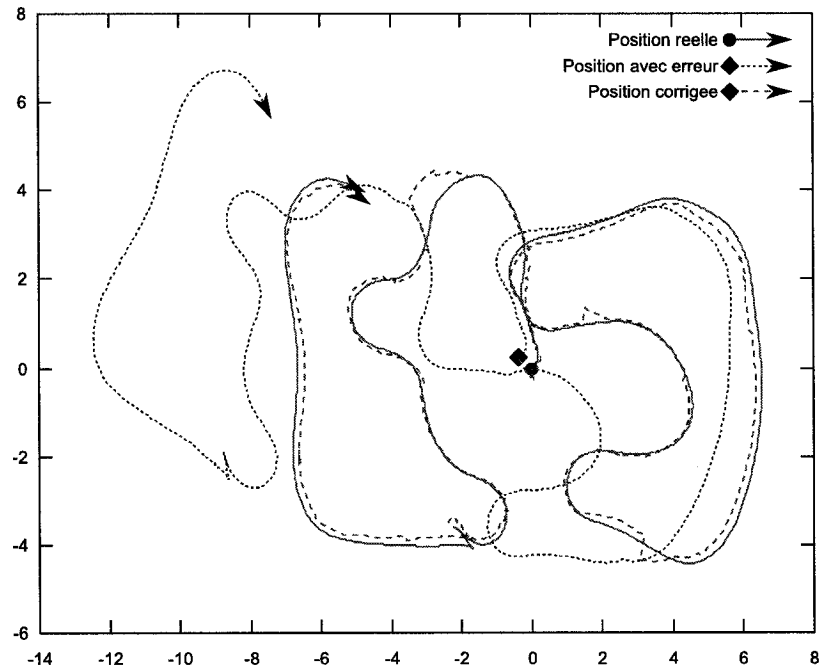


FIG. 4.6 – Résultats de la correction de la localisation par *IDC*

la correction (b).

Des tests, menés en expérimentation et en simulation, ont permis de vérifier le fonctionnement. La figure 4.6 montre les résultats en simulation dans un environnement structuré : une erreur de mesure est rajoutée aux données d'odométrie et les positions réelles, erronées et corrigées ont été dessinées. On voit que la position corrigée suit la position réelle alors que la position erronée diverge nettement.

4.3.1.3 Planification de chemin

Que ce soit pour effectuer une action après une détection ou pour se rendre sur une zone d'exploration localisée, les plates-formes doivent se déplacer vers une destination connue. Dans un environnement fait de pièces reliées par des couloirs et des portes, la trajectoire directe est rarement possible. Grâce à la connaissance

de l'environnement, la plate-forme peut planifier le chemin qu'elle empruntera pour se rendre à sa destination. La planification peut alors prendre plusieurs formes :

- une trajectoire exacte : la plate-forme prévoit à l'avance tous ses mouvements ; malheureusement, une telle approche ne permet pas d'éviter les obstacles inconnus ou dynamiques ;
- une trajectoire d'attraction : en associant à une trajectoire exacte pré-calculée la répulsion des obstacles détectés, la plate-forme peut ainsi les éviter ; comme la précédente, cette solution demande un effort de calcul important pour analyser les possibilités de trajectoires à chaque planification ;
- une séquence de points de passage : dans un environnement dont la topologie peut être décrite facilement, comme les intérieurs de bâtiments donc chaque pièce est reliée aux autres par des portes, on peut essayer de décrire le chemin à ce niveau : une séquence de pièces et de portes à traverser ; un deuxième niveau de contrôle permet le déplacement le long de cette séquence de points en évitant les obstacles.

La solution implantée pour la présente application cherche à utiliser le découpage en pièces et couloirs des environnements intérieurs, mais sans avoir à en décrire la topologie. En effet, on veut pouvoir utiliser seulement le plan des murs (comme celui de la figure 4.1) sans informations topologiques additionnelles. On construit un graphe dont les noeuds sont des points de la carte et dont les arêtes représentent les chemins en ligne droite entre ces points. Les arêtes sont étiquetées avec la distance séparant les deux points, et la planification de chemin revient à un parcours de graphe. La création, le maintien et la recherche dans le graphe sont implantés comme fonctionnalités au sein du module singleton de cartographie *MapClient*.

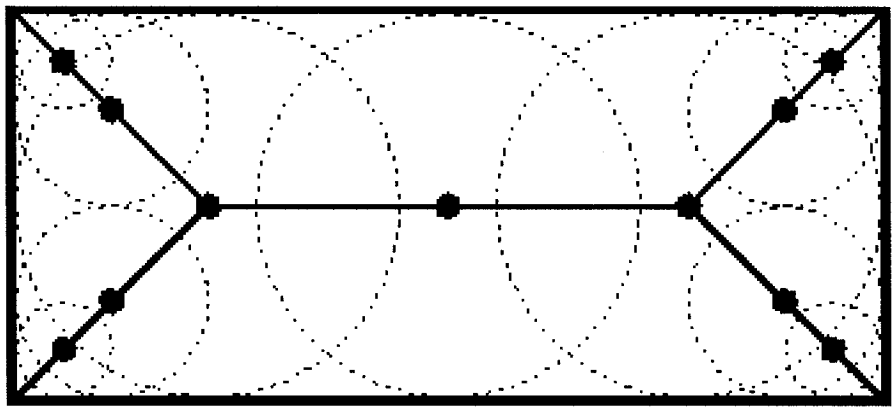


FIG. 4.7 – Squelettisation et cercles bi-tangents

Construction du graphe Ce graphe est construit par chaque plate-forme dès que la carte est connue (c'est-à-dire au lancement de l'application) et est enregistré. Cette construction demande plusieurs étapes.

- Pour permettre le passage d'une plate-forme mobile dans un espace, il faut s'assurer que la largeur de passage est supérieure à la largeur du robot. Une solution, artificielle mais simplifiante, consiste à épaissir tous les murs et les obstacles de la carte de la demi-largeur de la plate-forme ; ainsi, on peut se permettre de la considérer comme ponctuelle.
- La méthode d'intégration de la topologie de la carte de murs est inspirée du domaine du traitement d'image : on effectue une squelettisation[6] de l'espace libre (zone sans murs et sans objets épaissis). La squelettisation est une opération morphologique sur les formes : le squelette est l'ensemble des centres des cercles bi-tangents[7] aux contours de la formes. La figure 4.7 permet de visualiser sur un rectangle la définition du squelette. Pour construire le squelette, on utilise la méthode dite du *feu de prairie* : on enflamme les contours, on propage le feu à l'intérieur de la forme et on note comme points de squelette les points de

[6] *Skeletonizing* ou *Median Axis Transform*
[7] cercles tangents en deux points distincts du contour

rencontre de deux fronts du feu. La figure 4.8(a) montre le résultat sur la carte du laboratoire.

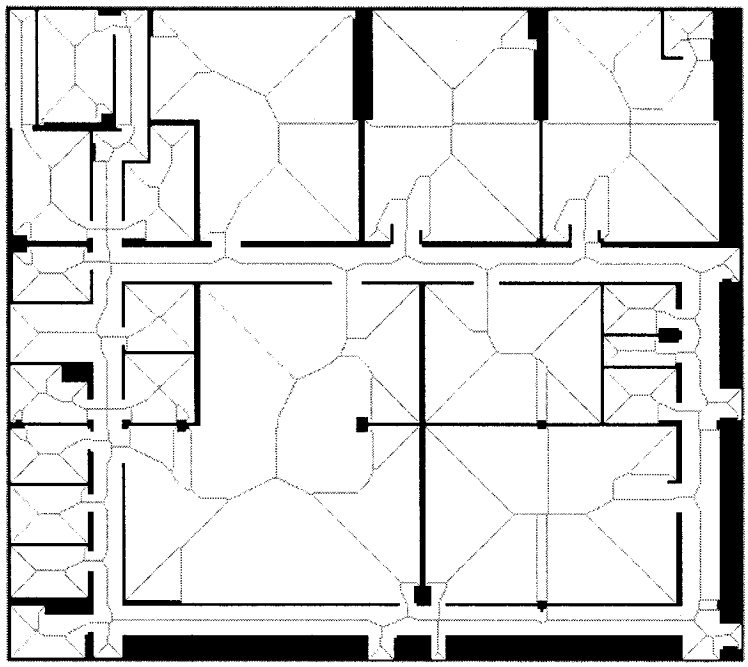
- On extrait les points caractéristiques du squelette, en ne gardant que les jonctions des arêtes. Ainsi, on diminue la taille du graphe tout en gardant la connectivité entre les points extraits. De plus, à partir de tous les points de l'espace libre de la carte, on peut atteindre un point caractéristique en ligne droite. La figure 4.8(b) montre les points caractéristiques extraits du squelette.
- On construit le graphe : si le chemin en ligne droite est possible entre deux points caractéristiques, l'arête entre les deux points est étiquetée avec la distance qui les sépare.

Recherche de chemin À chaque demande de planification (allocation d'une tâche ou requête de coût), le graphe est augmenté temporairement des points de départ et de destination. Les arêtes liant ces points aux points accessibles en ligne droite du graphes sont aussi créées. La planification de chemin s'identifie alors au problème classique de recherche de plus court chemin dans un graphe. Un algorithme efficace[8] est celui de *Dijkstra* (présenté dans [Dijkstra, 1971, p64-70]). On ne présentera pas ici son principe, puisqu'on le trouve dans tout livre d'introduction à la théorie des graphes. Appliqué avec comme source le point de départ et comme but le point d'arrivée, il permet de construire la séquence de points à suivre pour parcourir de façon optimale le chemin demandé.

4.3.1.4 Évitement d'obstacles

Dans un environnement connu, mais dynamique –c'est-à-dire comportant des obstacles inconnus ou se déplaçant–, le déplacement doit prendre en considération ces

[8] complexité en $O\left((m+n)\,ln\,(n)\right)$, où m est le nombre de noeuds et n le nombre d'arêtes



(a) Squelettisation pour la carte des locaux

(b) Points caractéristiques pour la carte des locaux

FIG. 4.8 – Squelettisation des zones libres

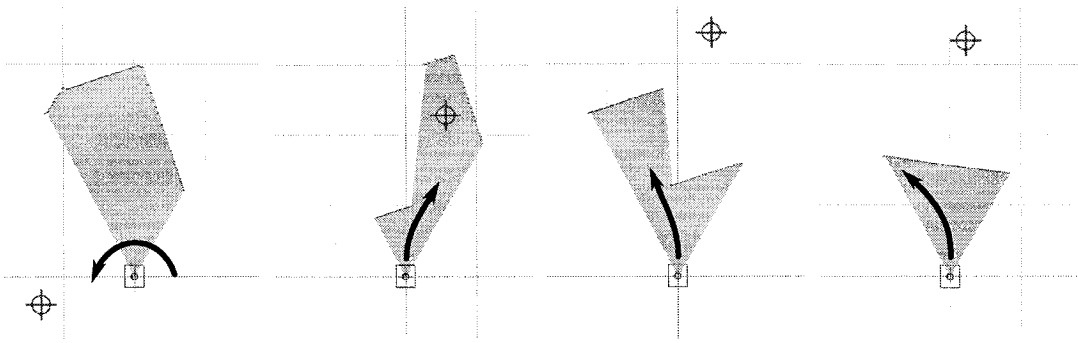


FIG. 4.9 – Commandes de déplacement point à point dans les différents cas d'obstacle

éléments inconnus. La prise en compte des obstacles se fait grâces aux différents capteurs (télémètres, sonars, infra-rouges, caméras...) et s'intègre en fin de chaîne de contrôle. L'utilisation de l'architecture de prototypage *Acropolis* a permis de séparer cette faculté d'évitement d'obstacles du reste de la chaîne de traitement, ainsi que de la découper en deux modules réutilisables : un déplacement point à point contournant les obstacles et un évitement local de collisions.

Déplacement point à point sécuritaire Lorsqu'on utilise la planification de chemin présentée à la section précédente, on obtient une séquence de points de passage, présumés reliables en ligne droite. Pour prendre en considération les obstacles inconnus lors du déplacement, un module de déplacement vers un point utilisant le télémètre laser pour les contourner a été développé : c'est le module *GotoSafe*. Ses entrées sont les positions actuelle[9] et de destination. Le principe de fonctionnement est le suivant :

– si l'angle entre l'axe de destination et la direction actuelle est supérieur à 90°, la

[9]incluant l'allure, i.e. la direction

plate-forme effectue une rotation sur elle-même (ex : figure 4.9(a)) ;
- si l'axe de destination entre dans le champ de vision du télémètre :
  - si aucun obstacle n'est détecté dans cette direction, la plate-forme effectue une rotation et un déplacement axial (ex : figure 4.9(b)) ;
  - si un obstacle apparaît dans la direction, et qu'une discontinuité supérieure à la largeur du robot est détectée dans un intervalle angulaire autour de la direction, la plate-forme se dirige vers cette discontinuité (ex : figure 4.9(c)) ;
  - si un obstacle apparaît et obstrue tout l'intervalle autour de la direction, la plate-forme choisit de contourner l'obstacle du coté qui présente le plus de champ libre (ex : figure 4.9(d)).

Enfin, les vitesses de déplacement sont proportionnelles à la distance soit au point de destination quand il n'y a pas d'obstacle, soit à l'obstacle que la plate-forme est en train d'éviter. La vitesse est seuillée par une valeur maximale définie pour chaque plate-forme.

Évitement local de collisions L'adaptation de la trajectoire ainsi réalisée ne suffit pas à éviter les collisions de proximité : en effet on s'intéresse principalement aux obstacles entre le centre du capteur et le point de destination. Si on se trouve déjà proche d'un obstacle, les manoeuvres décrites précédemment peuvent amener à des collisions. Il faut aussi prendre en compte les dimensions de la plate-forme : une rotation seule peut engendrer aussi une collision à cause de l'envergure de l'équipement. Un module d'inhibition des déplacement dangereux à été développé pour répondre à ce problème : *ObstAvoidance*. Il se place de manière transparente en entrée du module de pilotage des moteurs.

L'idée est de supprimer les commandes qui amèneraient à une collision du matériel avec un obstacle détecté. Pour cela, une grille d'occupation représentant l'espace

local autour de la plate-forme est maintenue. Cette grille est mise à jour par les données du télémètre et par le déplacement reporté par l'odométrie. De cette façon, la plate-forme maintient pendant un temps donné les informations sur les obstacles précédemment détectés, placés actuellement hors du champ d'acquisition du télémètre. L'inhibition se fait de manière indépendante sur les commandes de déplacement axial et de rotation : on simule leur application pendant un laps de temps supérieur à la constante de temps du contrôle et on inhibe l'une, l'autre ou les deux, suivant les points de collision.

4.3.2 Comportements

La définition des comportements se fait sous la formes de circuits de configuration *Acropolis*, dont le format est expliqué en 3.3.1.2. Les plates-formes peuvent réaliser un ou plusieurs des comportements nécessaires à la mission : le déplacement vers un point d'intérêt, l'exploration –dirigée ou non suivant l'application– et la segmentation en zones d'exploration. Une partie commune à ces circuits est aussi présente dans la configuration.

4.3.2.1 Partie commune

Les plates-formes utilisées sont pilotées par *Player* (cf. 3.2) ; plusieurs modules d'entrées/sorties pour *Player* sont donc présents dans la partie commune des circuits : *Odometry*, *Laser* et *Motor*.

Elle doivent aussi en tout temps connaître la carte de leur environnement et se localiser dans cette carte. Le module de cartographie *MapClient* (voir 4.3.1.1) est donc aussi présent dans la partie commune ainsi que le bloc de localisation et de correction par *IDC* présenté au 4.3.1.2.

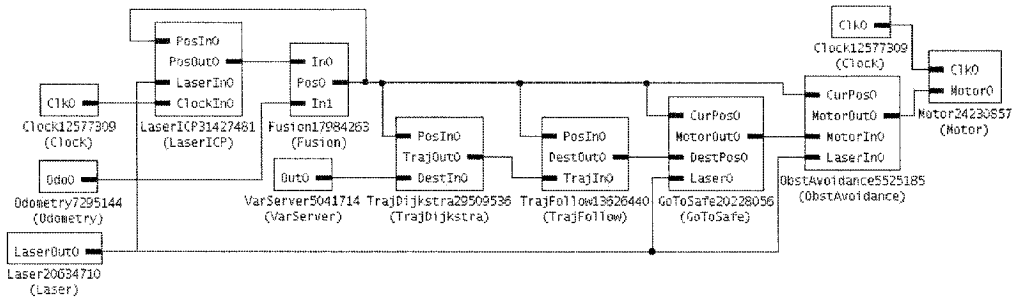


FIG. 4.10 – Circuit *Acropolis* pour le déplacement

Le module d'évitement de collisions local *ObstAvoidance* est associé à l'entrée du module de pilotage des moteurs, pour assurer l'intégrité des plates-formes.

4.3.2.2 Action : déplacement

Comme il a été défini en 4.1.2, la tâche d'action qui doit être effectuée après une détection d'un point d'intérêt est simulée par le déplacement d'une plate-forme capable de l'action vers le point en question. La figure 4.10 montre le circuit complet réalisant ce comportement (incluant les parties communes). Une version élargie des figures représentant les circuits de configuration se trouve en annexe I.

Les options de spécialisation de la tâche que sont les coordonnées du point d'intérêt sont traitées par un module d'utilisation des paramètres (*VarServer*, cf 3.3.2.3). La sortie fournie est utilisée par le module de planification de chemins (*TrajDijkstra*, cf 4.3.1.3), dont la séquence de points calculée est traitée par un module de suivi de chemin (*TrajFollow*). Celui-ci met à jour sa sortie en y présentant le prochain point dès que la position courante est proche[10] du point actuel. Les points sont atteints

[10]la distance de passage près des points est paramétrable, selon la taille du robot

grâce aux deux modules de déplacement sécuritaire *GotoSafe* et d'évitement local de collision *ObstAvoidance*.

Lorsque le dernier point de la séquence est atteint, *TrajFollow* émet l'évènement « *ok* ». Cet évènement permet de prévenir l'équipe de la fin de l'action, en libérant le jeton de la place associée à cette tâche.

4.3.2.3 Exploration non dirigée

Le comportement d'exploration doit mettre à jour la carte avec les zones couvertes, commander les mouvements de la plate-forme et détecter les points d'intérêt. Les deux premières tâches sont effectuées en grande partie par le module d'exploration *Explorer*, alors que la détection se fait à part. La figure 4.11 montre le circuit de configuration dans le cas de l'exploration non dirigée.

Sur la figure 4.11, on voit un module *Fiducial*, qui est un module d'entrée *Player*, utilisé seulement en simulation : cette entrée est une simulation de détection de repères implantée dans l'environnement de simulation *Stage*. Lors des tests expérimentaux, le module de détection de repères a pu être remplacé par une caméra omnidirectionnelle et un module de détection par segmentation[11] colorimétrique implanté par Vincent Zalzal pour son projet de maîtrise [Zalzal, 2005]. Les détections sont envoyées au module *FidPos* qui émet un évènement « *found* » qui génère un jeton dans la place d'action. *FidPos* reçoit aussi par message les détections de l'ensemble de l'équipe de façon à ne pas prendre en compte un point déjà détecté.

L'exploration proprement dite est gérée par le module *Explorer*. Un module d'horloge active ce module toutes les 4 secondes. Pour décider de la direction de naviga-

[11] les points de repères utilisés sont des cylindres dont les couleurs sont connues

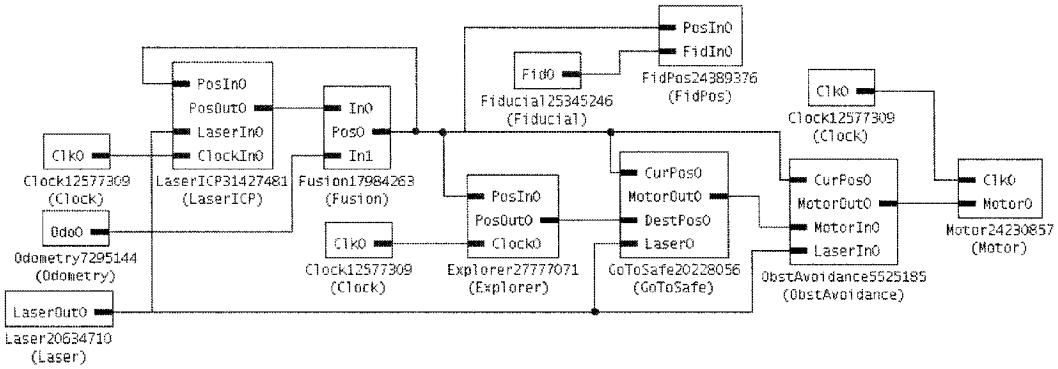


FIG. 4.11 – Circuit *Acropolis* pour l'exploration non dirigée

tion, le module analyse les cases de la grille d'occupation sur un cercle légèrement plus grand que la portée de la détection. Le module dirige alors la plate-forme vers :

- la case non explorée et accessible dont la direction est la plus proche de la direction actuelle si il y en a une,
- la case dont la dernière exploration est la plus ancienne sinon.

Pour effectuer le déplacement, les coordonnées de la zone sont fournies à *GotoSafe*, qui génère les commandes envoyées aux moteurs de façon sécuritaire via le module d'évitement local de collision. En parallèle, *Explorer* marque les zones comme explorées dans un rayon égal à celui de la détection.

4.3.2.4 Exploration dirigée

L'exploration dirigée consiste à se déplacer vers la zone attribuée par le gestionnaire d'équipe, puis de l'explorer. Ce comportement est en fait une fusion des comportements de déplacement et d'exploration non dirigée. La figure 4.12 montre le circuit associé. On peut voir que le module *GotoSafe* voit son entrée de destination reliée aux sorties du module de suivi de chemin et du module d'exploration, *Explorer2*.

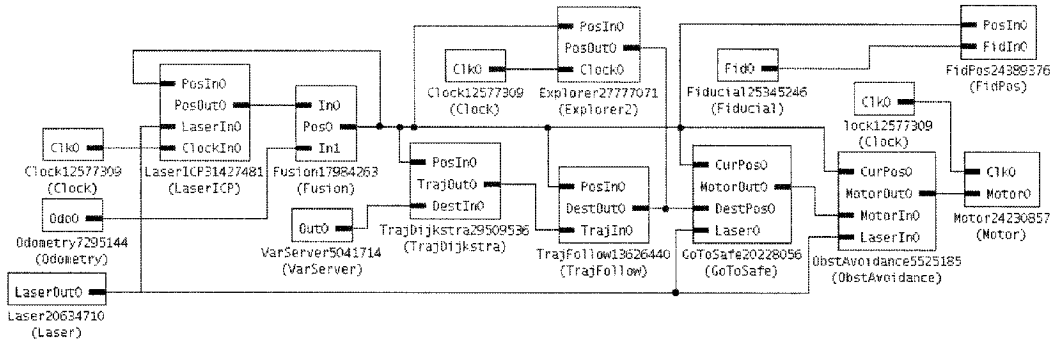


FIG. 4.12 – Circuit *Acropolis* pour l'exploration dirigée

Pour éviter un comportement erratique de la plate-forme, ces deux modules ne doivent pas fonctionner en même temps. Le module d'exploration diffère quelque peu de son prédécesseur :

- il reste inactif tant qu'il ne reçoit pas le message d'arrivée de *TrajFollow* ;
- la navigation s'arrête quand il n'y a plus de zones non non explorées et accessibles, l'évènement « *search_ok* » est alors émis.

Ce comportement est donc en fait composé de deux phases distinctes[12] : déplacement puis exploration, toutes deux accompagnées de la détection des points d'intérêt.

4.3.2.5 Découpage en zones d'exploration

Ce comportement est purement calculatoire et peut être implanté sur une ou plusieurs des plates-formes membres de l'équipe, ou sur un membre spécial, seulement logiciel (exécuté par exemple sur le même hôte que le gestionnaire d'équipe).

[12] l'amélioration discutée en 3.4.2.2, concernant une gestion locale des tâches, pourrait permettre de simplifier la définition de ce comportement en le scindant en deux comportements séquentiels

Le découpage en zones d'exploration consiste à analyser la carte actuelle et à fournir à l'équipe une liste de zones non explorées. Ce découpage est effectué par un module *MapSegment* qui effectue une opération de segmentation par l'algorithme de *FloodFill* sur les zones non explorées. Cet algorithme, tiré du domaine du traitement d'images, permet de créer des régions de pixels contigus ayant une même propriété. Dans notre cas, les pixels sont les cases de la grille d'occupation et on se restreint aux cases non explorées.

Si le nombre de régions détectées est supérieur au nombre demandé *nb_zones* (qui est une option de spécialisation de la tâche), seulement les plus grandes sont retenues. Si ce nombre est inférieur, les plus grandes sont coupées récursivement en moitié : verticalement ou horizontalement suivant la disparité selon ces axes. Finalement, la position de la case la plus proche du point médian de chaque zone est utilisée comme destination pour l'exploration dirigée. Les différents évènements sont alors émis :

- N évènements de type « *flush_zones* » pour libérer les jetons représentant les anciennes zones ;
- N évènements de type « *zone* » représentant les nouvelles zones à explorer ;
- 1 évènement de type « *segment_ok* » informant de la fin de la segmentation.

Conclusion

Le déploiement d'une telle application a demandé d'une part la description de la mission au niveau de l'équipe mais aussi le développement des comportements associés aux tâches nécessaires à son aboutissement. Le cheminement effectué au cours de ce chapitre illustre ce qu'un développeur d'applications multi-robot utilisant l'architecture présentée aura à effectuer pour arriver à ses fins. Cette application

permet aussi de tester le fonctionnement de l'architecture de contrôle dont le développement est la motivation principale du présent projet. L'ensemble des éléments relatifs à ces tests et les résultats obtenus sont présentés dans le chapitre suivant.

CHAPITRE 5

RÉSULTATS ET ANALYSE

Le présent projet a mené au développement d'une architecture de contrôle et au déploiement d'une application de celle-ci. Ce chapitre vient conclure ce développement en présentant l'état d'aboutissement du projet et une partie des tests qui ont servi à sa validation. La première partie traitera des outils utilisés pour mener les tests ainsi que du fonctionnement des comportements développés pour l'application d'illustration décrite au chapitre précédent. Deux sections seront ensuite consacrées à la présentation de résultats des tests de l'architecture effectués dans le cadre de cette application, avec des équipes d'abord restreintes puis comportant un plus grand nombre de membres. Enfin, un bilan sur le travail accompli et les perspectives de développement envisagées pour le futur seront présentés dans la dernière section.

5.1 Généralités

5.1.1 Niveaux d'expérimentation/simulation

Pour tester le fonctionnement d'une telle architecture, l'application proposée –le *Search-and-Rescue*– nécessite évidemment l'implication d'une équipe de plusieurs plates-formes robotiques. Le matériel présent sur les plates-formes du laboratoire ne permet pas à celles-ci de toutes se qualifier pour cette application (taille, présence d'un télémètre ..). Une seule plate-forme est équipée adéquatement : un robot roulant *ATRV-Mini* distribué par *iRobot*, embarquant un télémètre laser *SICK*

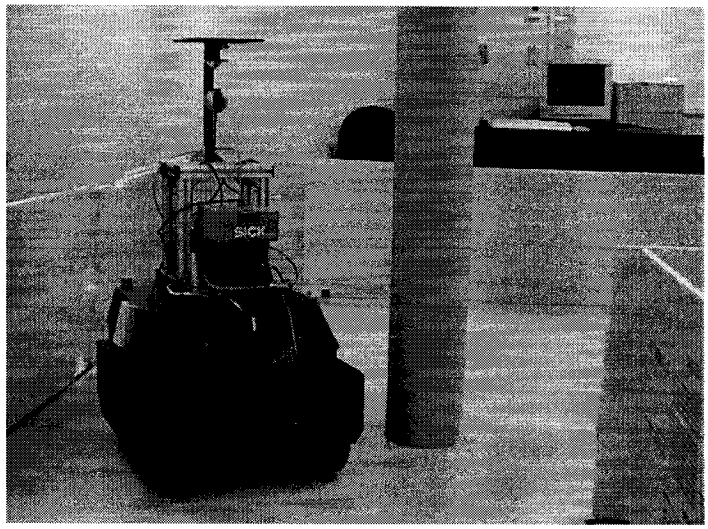


FIG. 5.1 – L'*ATRV-Mini* dans son environnement de test

LMS200 et une caméra omnidirectionnelle[1], représenté sur la photo de la figure 5.1. Celle-ci a donc servi à prouver le fonctionnement de chacun des comportements.

Pour les tests en équipe de l'architecture complète, l'environnement de simulation *Stage*, fourni comme module de *Player*, permet donc de simuler un nombre quelconque de plates-formes. On utilise dans cet environnement des modèles de la plate-forme de test *ATRV-Mini*, dont on a pu vérifier la validité en appliquant les comportements développés pour la mission. Malheureusement, la simulation dans *Stage* d'un nombre important de plates-formes, utilisant des capteurs gourmands en ressources comme les télémètres laser, pose des problèmes de capacité de calcul et a mis au jour certaines failles[2] dans le simulateur. De plus, les comportements développés dans *Acropolis* demandent eux-aussi d'importantes capacités de calcul. Cet ensemble de contraintes limite la composition des équipes : au delà de 5 ou 6 plates-formes, le simulateur ne fournit plus assez de données pour assurer un comportement normal des robots.

[1] composée d'une caméra *Logitech QuickCam Pro 4000* et d'un miroir hyperboloïdal

[2] un récent changement de l'architecture de *Player* corrige ces failles, mais le portage d'*Acropolis* à cette nouvelle version n'est pas finalisé à l'heure actuelle

Pour pouvoir tester de manière plus poussée le gestionnaire d'équipe, un simulateur simpliste a été développé. Il remplace le *proxy* qui d'ordinaire effectue le lien entre l'équipe et *Acropolis*. Chaque instance de ce programme peut lister ses capacités en terme de tâches, répondre aux requêtes de coût, accepter les affectations de tâches et finalement émettre les évènements associés.

Finalement, une interface usager permet d'émettre des évènements à la volée. Ce programme prend la forme d'une console interactive invitant l'usager à définir des évènements à envoyer. Un mode « scripté » permet aussi de définir un scénario temporel d'envoi de messages.

5.1.2 Test expérimental des comportements

Parallèlement au développement du système de coordination et à l'adaptation des outils présents, le projet a nécessité le développement des différents comportements utilisés par l'application utilisée pour l'illustrer. Le processus de développement de ces comportements a été formé d'une alternance de développements en simulation et de validations en expérimentation. En effet, et ce grâce à l'architecture modulaire d'*Acropolis*, chaque module a pu être développé séparément. Une attention particulière a été portée à la validation expérimentale des modules qui présentaient des possibilités de divergence avec la simulation :

- le module de relocalisation par *IDC* (cf 4.3.1.2) : un suivi des positions calculées a été effectué pour s'assurer de son fonctionnement hors de la simulation ;
- le module d'évitement local de collisions (cf 4.3.1.4) : l'insertion de ce module à un comportement de contrôle au clavier a permis de vérifier son efficacité face à des obstacles réels
- le module de déplacement point à point sécuritaire (cf 4.3.1.4) : son fonctionne-

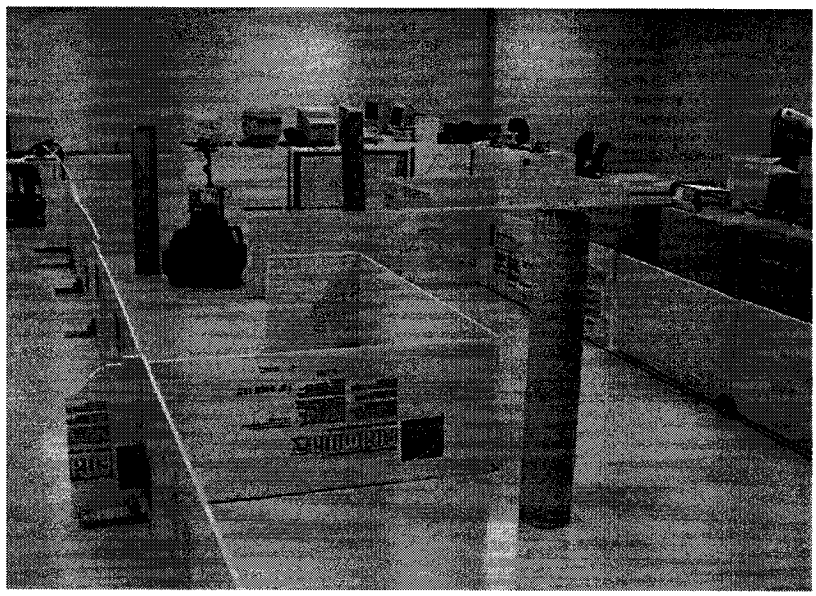


FIG. 5.2 – Environnement de test avec la plate-forme *ATRV-Mini*

ment a été testé grâce à une entrée interactive des coordonnées de destination et le placement d'une série d'obstacles placés sur les trajectoires.

Finalement, les comportements ont été testés dans leur globalité, en prenant soin de valider les mouvements liés aux comportements comprenant la partie de déplacement (déplacement pour action et exploration dirigée). La photo de la figure 5.2 montre la plate-forme *ATRV-Mini* dans l'environnement de test : les murs sont faits de plaques dont la hauteur est juste supérieure à la position du télémètre laser, de façon à qu'ils soient pris en compte par la plate-forme sans géner l'observation. La figure 5.3 est le plan des murs de cet environnement ; cette carte est celle connue par la plate-forme. La carte ne référence pas les obstacles (les cylindres colorés visibles sur la photo de la figure 5.2).

La difficulté pour montrer les résultats d'expérimentation dans un document écrit ont poussé à produire des séquences vidéos, consultables sur le site Internet du *GRPR* (à l'adresse *http ://www.ai.polymtl.ca/projet_rg/*). Ces vidéos présentent

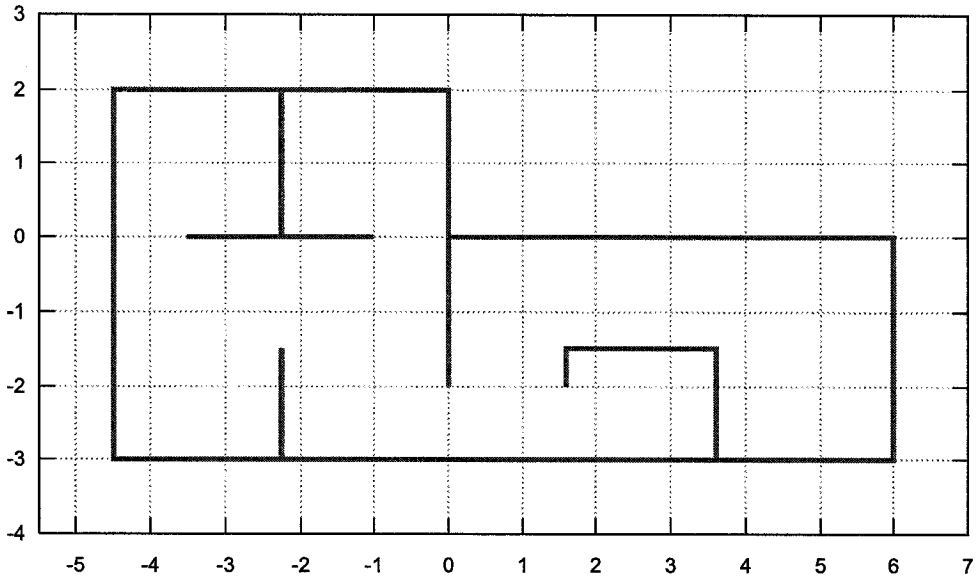


FIG. 5.3 – Carte des murs de l'environnement de test

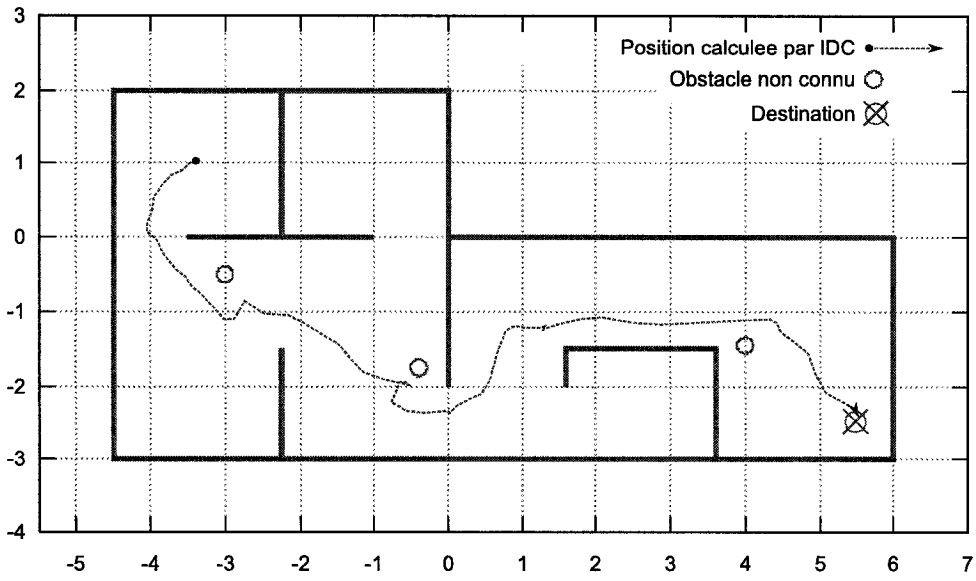


FIG. 5.4 – Déplacement de la plate-forme lors du trajet de $(-3.5, 1)$ à $(5.5, -2.5)$

la plate-forme utilisant le comportement de déplacement d'un point à un autre pour se rendre à des points précis de la carte. Pour réaliser ces test, le système de coordination est configuré pour la mission de *Search-and-Rescue* avec exploration non-dirigée et l'équipe se réduit au seul robot *ATRV-Mini*. Le lanceur d'évènements permet alors de désigner des cibles à traiter. Pendant les tests, la position calculée par le système de relocalisation a été enregistrée au cours du temps. La figure 5.4 montre la séquence des positions lors d'un déplacement de la position de départ $(-3.5, 1)$ à la position de la cible $(5.5, -2.5)$. Les cercles sur la figure montre les positions des obstacles lors de ce test.

5.2 Assignation en équipe restreinte

Pour se rendre compte du fonctionnement de l'assignation lors des missions, il peut être intéressant de suivre l'affectation de chaque membre de l'équipe. Néanmoins, pour une équipe composée de nombreux robots, il est difficile de représenter l'activation de chaque comportement sur chaque plate-forme. C'est pour cela que des tests en équipe réduite ont été menés : on va se limiter dans un premier temps à trois équipiers. Un protocole de test en équipe élargie est discutée à la section 5.3.

5.2.1 Protocole

Ces expérimentations utilisent l'environnement de simulation *Stage* dans lequel trois plates-formes *ATRV-Mini* sont simulées. Ce qui nous intéresse ici est l'affectation des différents membres de l'équipe aux tâches nécessaires à la mission. La mission étudiée ici est celle utilisant l'exploration non dirigée. Pour pouvoir comparer les effets de la composition de l'équipe (capacités respectives des membres), on a besoin d'un scénario reproductible. Dans cette optique, les points d'intérêt ne

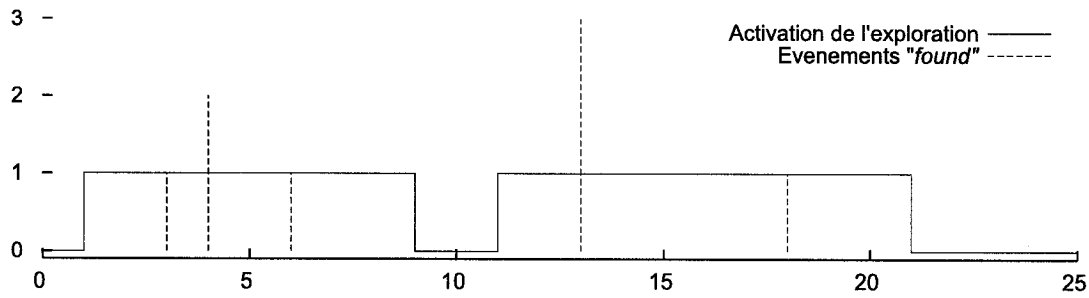


FIG. 5.5 – Ligne de temps des évènements

seront pas détectés par les membres mais on simulera leurs détections grâce l'outil d'envoi d'évènements : un script permet de répéter un scénario d'évènements de détection suivant une ligne de temps précise.

La figure 5.5 montre la ligne de temps associée au script utilisé dans ces tests :

- la courbe pleine montre la période d'activation de l'exploration : activée au lancement de l'application, mise en pause à $t = 9s$ par l'envoi de l'évènement « *pause* » et réactivée à $t = 11s$ par l'envoi de « *resume* » ;
- les barres verticales pointillées indiquent le nombre des détections simulées par l'envoi d'évènements « *found* »(par exemple 3 détections à $t = 13s$).

Plusieurs compositions d'équipe ont été testées :

- les trois plates-formes capables des deux tâches ;
- deux capables de l'exploration et une du déplacement ;
- deux capables des deux tâches, une capable du déplacement ;
- une capable de l'exploration, une du déplacement et une des deux.

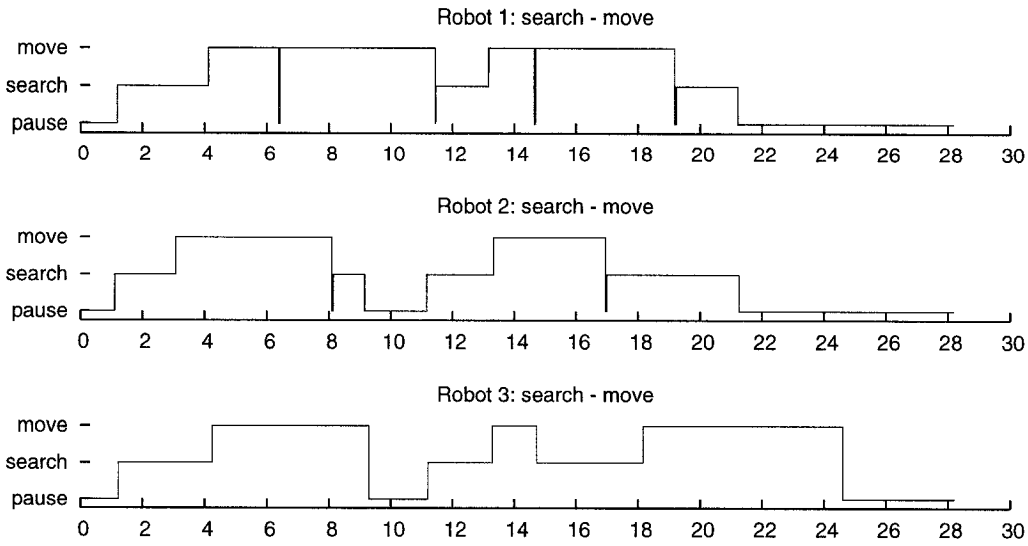


FIG. 5.6 – Assignation en équipe restreinte - Configuration 1

5.2.2 Résultats temporels

La première configuration étudiée est une configuration homogène, où les trois plates-formes sont capables des deux tâches nécessaires à la mission : explorer et se déplacer sur les points d'intérêt. La figure 5.6 montre l'état d'activation des trois membres dans le temps. Dès le lancement de l'application, les trois plates-formes sont affectées à la tâche de recherche, jusqu'à l'apparition de la première détection ($t = 3s$). Le deuxième agent est alors affecté à cette tâche tandis que les deux autres continuent l'exploration. Il est intéressant de remarquer que la première plate-forme complète une tâche de déplacement vers $t = 6.4s$, et est immédiatement réaffecté à une nouvelle tâche qui se trouvait en attente[3]. La réaffectation se traduit sur le graphe d'activité par un court passage de son état en pause (car la mesure d'activité se fait au niveau de l'assignation, mais la mise en pause n'est pas effective au niveau de la plate-forme qui, elle, commute directement d'une action à l'autre).

[3]le jeton existait dans la place de l'action mais la tâche était non affectée

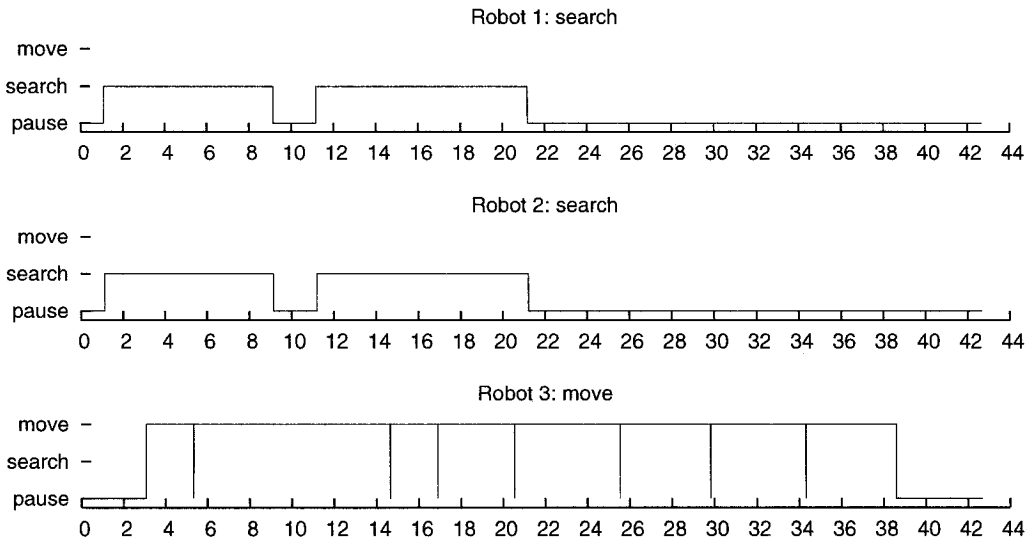


FIG. 5.7 – Assignation en équipe restreinte - Configuration 2

On peut aussi se rendre compte que la mise en pause du système pendant l'intervalle $[9s, 11s]$ stoppe bien les activités de recherche mais laisse s'effectuer les tâches de déplacement en cours.

La figure 5.7 montre les états d'activation pour la seconde configuration, utilisant des plates-formes non-polyvalents : deux capables de l'exploration et une capable du déplacement. Les deux premiers agents sont affectés en permanence à la tâche d'exploration pendant les périodes d'activité. L'agent 3, le seul capable des déplacements, se voit alors affecté à l'ensemble des points d'intérêts séquenciellement. Il retourne en état de pause après avoir effectué les 8 tâches qui lui sont affectées.

Dans la troisième configuration, dont les états d'activation sont présentés sur la figure 5.8, un des agents est seulement capable des déplacements, alors que les deux autres sont capables des deux tâches. Cette configuration permet de mettre en lumière la capacité du système à maximiser l'occupation des membres : si une seule tâche de déplacement est requise ($t = 3s$ et $t = 18s$), elle sera assignée au

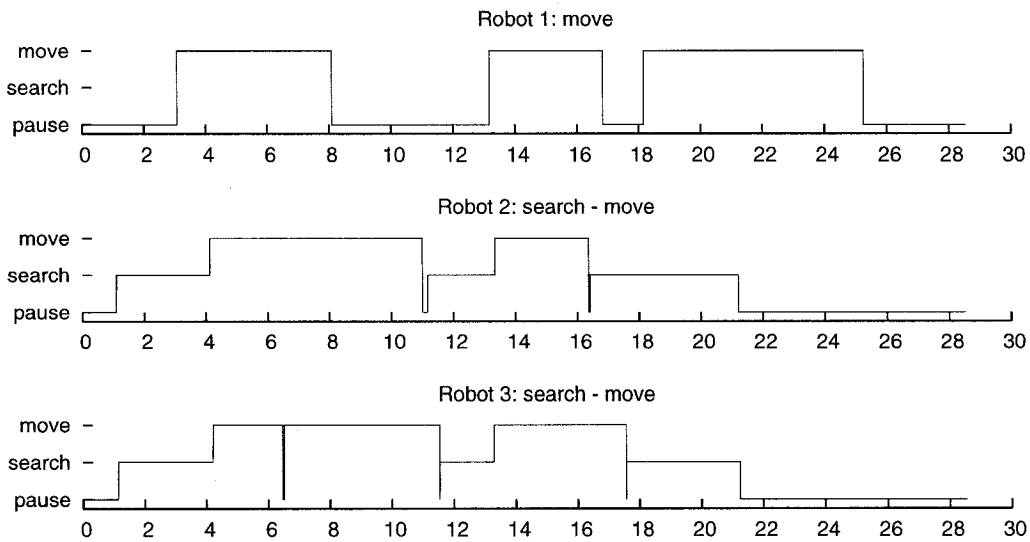


FIG. 5.8 – Assignation en équipe restreinte - Configuration 3

robot incapable de l'exploration : de cette manière l'exploration n'est pas diminuée par l'arrivée de cette tâche. On note aussi que lorsqu'une tâche de déplacement est en cours sur un des robots polyvalents, elle se poursuit même s'il serait alors possible d'optimiser l'assignation en affectant le premier robot à cette tâche quand il est inactif (ex : $t = 8s$) ; ce comportement est un paramètre de configuration de la mission, la tâche de déplacement étant notée comme non-sécable.

La figure 5.9 montre les états d'activation pour les membres de la dernière configuration : les trois plates-formes sont différentes, l'une ne pouvant qu'explorer, la seconde que se déplacer et la dernière étant capable des deux. Alors que le premier membre est affecté en tout temps à l'exploration et que le second reçoit en priorité les affectations des tâches sur les points d'intérêt, le dernier est affecté à l'exploration si il n'y a pas de points d'intérêt en souffrance.

D'une manière générale, on peut s'apercevoir que les contraintes d'assignation sont respectées, dans la mesure des ressources disponibles : les tâches d'action sur les

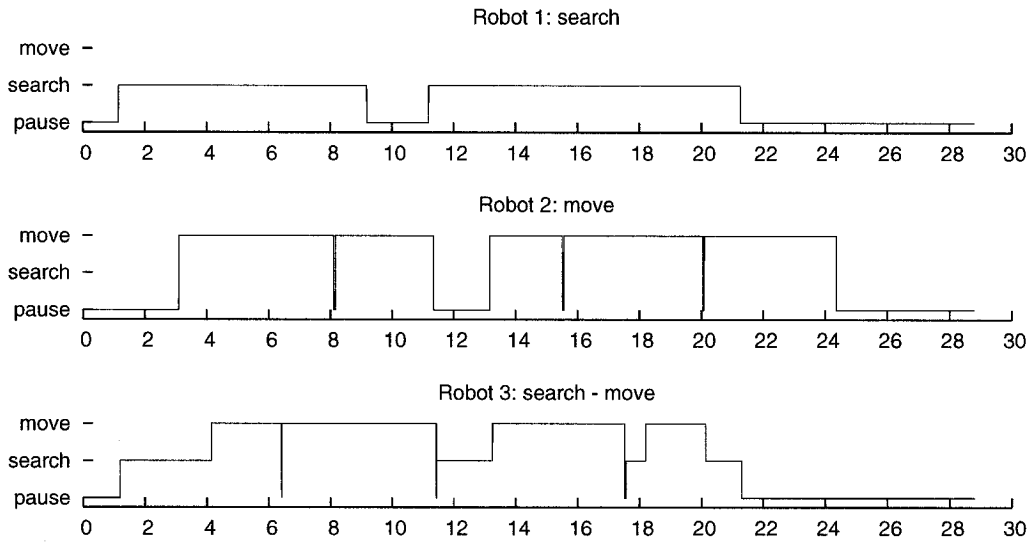


FIG. 5.9 – Assignation en équipe restreinte - Configuration 4

points d'intérêt sont affectées en priorité dès qu'elles apparaissent et la tâche d'exploration est affectée à tous les agents capables et disponibles.

En comparant la figure 5.7 aux trois autres, on voit nettement que le fait de n'avoir qu'un seul membre capable des actions sur les points d'intérêt pénalise l'équipe quand la fréquence des détections est élevée : la mission se termine à $t = 38s$ avec cette configuration alors qu'elle se termine aux environ de $t = 25s$ dans les autres configurations.

5.2.3 Résultats instantanés

Un deuxième point intéressant à vérifier, après le respect des contraintes et la maximisation de l'utilisation des agents dans le temps, est l'optimalité de l'assignation des différentes tâches aux agents. Pour cela, on prend une photographie instantanée des coûts estimés par les agents pour chacune des tâches à affecter et on observe l'affectation que le système calcule. Le respect des contraintes ayant déjà été dis-

	Robot 1	Robot 2	Robot 3
1 : *search*	**0**	0	**0**
2 : *move*	8.41	**3.12**	7.32

TAB. 5.1 – Coûts reportés par les plates-formes - 1 point d'intérêt

	Robot 1	Robot 2	Robot 3
1 : search	0	0	0
2 : move	5.00	5.00	**7.07**
3 : move	**2.00**	8.60	5.39
4 : move	10.30	**4.00**	6.40

TAB. 5.2 – Coûts reportés par les plates-formes - 3 points d'intérêt

cuté, on va se placer dans le cas où tous les membres sont capables de toutes les actions (configuration 1).

Une première situation présente à l'équipe un seul point d'intérêt détecté et la tâche d'exploration active. Le tableau 5.1 présente les coûts que les trois plates-formes ont estimés pour les deux tâches présentes. Le coût de la tâche d'exploration est estimé nul dans le cas de l'exploration non dirigée alors que le coût du déplacement est calculé en divisant la distance à parcourir par la vitesse du robot.

Pendant l'assignation de ce cas, la tâche d'exploration ne sera affectée qu'à la troisième passe de l'algorithme d'assignation (cf 2.4.3.2) : les agents restant libres après l'assignation des tâches prioritaires –ici le déplacement– servent à compléter les tâches dont la borne supérieure d'assignation –ici $+\infty$– n'est pas atteinte. Le résultat de l'assignation est le suivant : le robot 2 est affecté à la tâche de déplacement, alors que les deux autres sont affectés à l'exploration. On peut voir facilement qu'il s'agit là de l'assignation optimale : la tâche de déplacement doit être assignée à un des robots, et le robot 2 donne la plus petite estimation du coût de cette tâche.

La situation suivante fait apparaître trois détections de points d'intérêt, la tâche

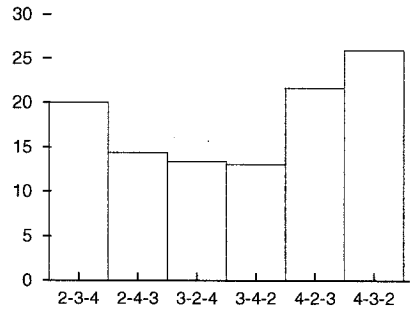


FIG. 5.10 – Sommes des coûts pour les différentes assignations

d'exploration étant toujours active. Le tableau 5.2 donne les coûts reportés par les trois plates-formes pour les tâches associées et fait apparaître l'assignation calculée. La figure 5.10 montre les coûts pour toutes les assignations (le triplé en abscisse représente l'indices tâches assignées respectivement aux trois robots). On voit que l'assignation calculée (3-4-2) est bien celle qui minimise la somme totale.

Des vérifications similaires ont été faites pour un nombre plus grand de tâches et il en ressort que le gestionnaire commence par assigner le sous-ensemble qui représente le coût total minimal. Les autres tâches seront assignées dès la libération de l'un des membres. Ce comportement, bien que logique dans le cas de cette mission (on préfère traiter les points d'intérêt proches en premier), peut amener à un comportement non voulu : si de nouvelles tâches ayant des coûts relativement faibles apparaissent constamment, l'assignation d'une tâche ayant un coût plus élevé pourrait être indéfiniment reportée.

5.3 Assignation en équipe élargie

Si la simulation utilisant l'architecture *Acropolis* et le simulateur *Stage* doit se limiter à quelques membres, pour des raisons matérielles, le gestionnaire d'équipe développé dans le présent projet peut quant à lui gérer un nombre quelconque de

robots. Pour vérifier le fonctionnement de celui-ci avec des équipes plus importantes, on peut utiliser le simulateur simpliste qui remplace l'ensemble de la couche locale (*proxy*, *Acropolis*, *Player*).

5.3.1 Protocole

Le simulateur utilisé est très basique : les comportements sont réduits à leur plus simple expression. Les agents simulés répondent aux requêtes de coûts, simulent l'action des comportements pendant la durée de la tâche et émettent les évènements associés aux fins de tâches. De plus, pour les déplacements et l'exploration dirigée, ils maintiennent la position des plates-formes et les coûts sont calculés en fonction de la distance euclidienne à la destination. Le protocole est semblable à celui utilisé pour les tests en équipe restreinte : une fois tous les membres de l'équipe démarrés, et le gestionnaire lancé, on utilise l'outil d'envoi d'évènements pour simuler les détections. Pour les deux missions, utilisant les deux modes d'exploration, différentes équipes ont eu à répondre à différents scénarii.

Trois compositions d'équipes différentes ont été utilisées, dans les versions associées aux deux modes exploratoires :

- la première équipe est composée de 5 plates-formes réalisant la tâche d'exploration (*explorateurs*), 5 réalisant les tâches de déplacement et d'action (*actionneurs*) et 5 capables des deux tâches (*polyvalents*) ;
- la seconde comprend 15 *explorateurs*, 5 *actionneurs* et 15 *polyvalents* ;
- la dernière utilise 15 plates-formes de chaque type pour obtenir un total de 45 membres.

Dans les tests en mode d'exploration dirigée, un membre supplémentaire est associé à l'équipe pour permettre la segmentation en zones d'exploration. Ce membre n'est

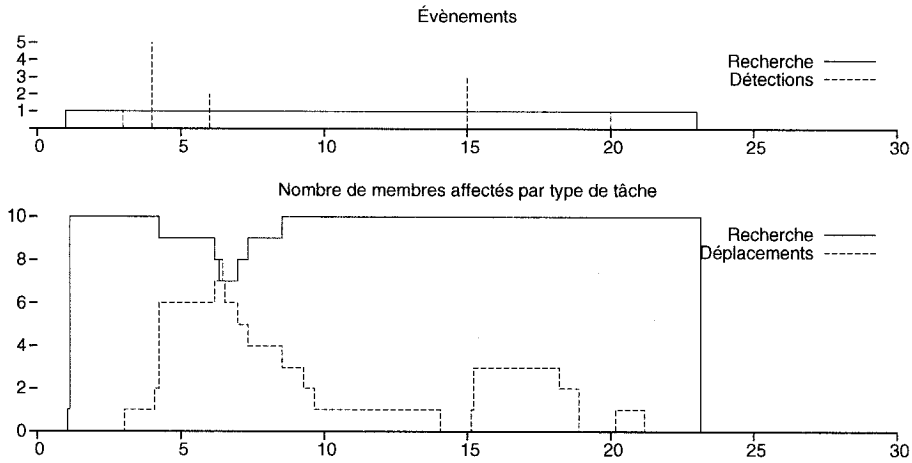


FIG. 5.11 – Résultats d'affectation, exploration non-dirigée, configuration 5-5-5, scénario 1

pas simplement simulé car le résultat de son action conditionne le fonctionnement de la mission. Il utilise toutes les couches de l'architecture normale et c'est le comportement développé dans *Acropolis* qui fournit la capacité de segmentation (la carte des murs du laboratoire est utilisée pour effectuer la segmentation, même si cela n'est pas pris en compte dans la simulation des autres plates-formes).

5.3.2 Résultats temporels

Les quatre premiers tests présentés correspondent à ceux menés avec la première équipe : 5 explorateurs, 5 actionneurs et 5 plates-formes polyvalents.

Le premier scénario de détection utilisé est assez semblable à celui de la section précédente, à la différence du retrait de la période de pause et du nombre de détections simultanées envoyées : jusqu'à 5 détections au temps $t = 4s$. Les détections correspondent à des points d'intérêt assez proches, dont la durée de déplacement ne dépasse pas 10 secondes. La figure 5.11 montre le scénario et le nombre d'agents af-

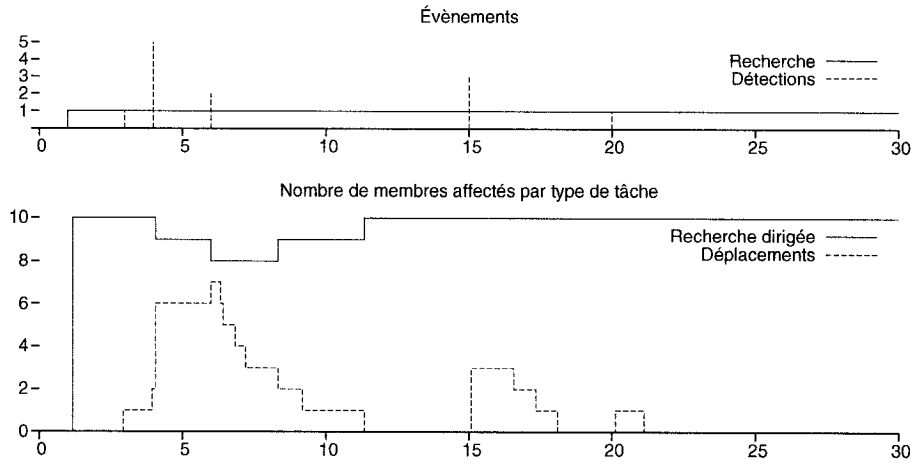


FIG. 5.12 – Résultats d'affectation, exploration dirigée, configuration 5-5-5, scénario 1

fectés à chaque type de tâche dans le temps pour le mode d'exploration non-dirigée. Il s'agit du nombre d'agents affectés à la tâche unique d'exploration et du nombre d'agents affectés à l'ensemble des tâches de déplacement. Tant que le nombre de tâches de déplacement demandées le permet (inférieur au nombre d'actionneurs : 5), tous les explorateurs et toutes les plates-formes polyvalents sont assignés à l'exploration. Quand cela est nécessaire, le système réaffecte des plates-formes polyvalents aux tâches de déplacement sur-numéraires.

La figure 5.12 montre la même équipe confrontée au même scénario de détections, mais utilisant le mode d'exploration dirigée. La tâche de segmentation, dont la période d'activation a été fixée à 15 secondes et à laquelle est demandée 15 régions par segmentation, n'est pas représentée, mais apparaîtrait sous la forme d'une impulsion au temps $t = 1s$, $t = 16s$, etc... On voit que le comportement général du système d'assignation est globalement le même que dans le cas de l'exploration non dirigée.

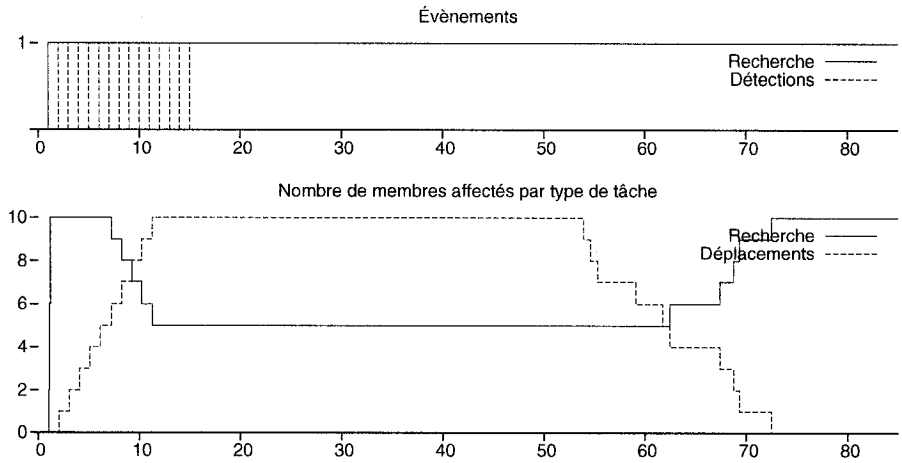


FIG. 5.13 – Résultats d'affectation, exploration non-dirigée, configuration 5-5-5, scénario 2

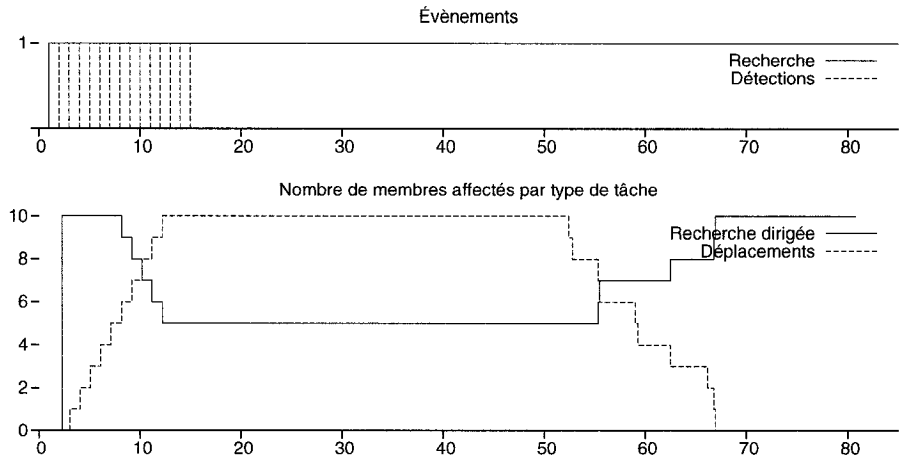


FIG. 5.14 – Résultats d'affectation, exploration dirigée, configuration 5-5-5, scénario 2

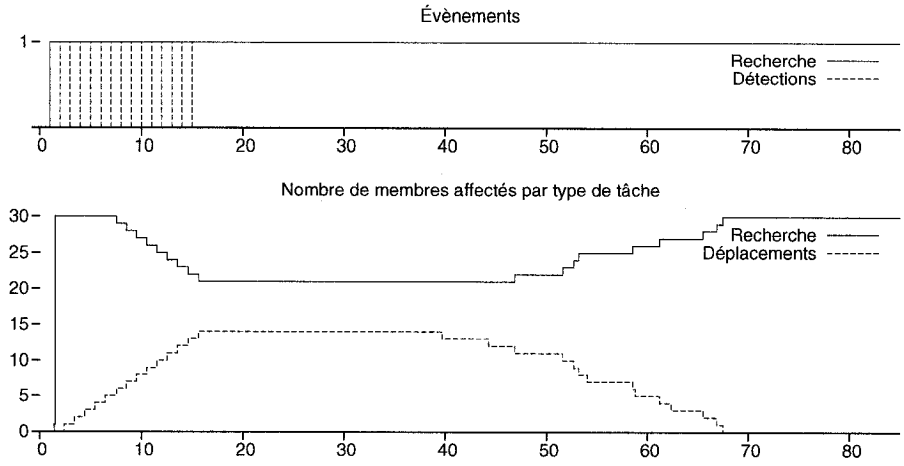


FIG. 5.15 – Résultats d'affectation, exploration non-dirigée, configuration 15-5-15, scénario 2

Les mêmes équipes sont utilisées pour le test suivant, dont le scénario est composé d'une série de 14 détections émises l'une après l'autre avec un intervalle d'une seconde entre chaque. Les figures 5.13 et 5.14 montrent respectivement les nombres d'assignations avec ce scénario dans le mode d'exploration dirigée et non-dirigée. Les détections de ce scénario sont plus lointaines et la durée de déplacement vers les points associés est plus longue que pour le premier scénario. L'assignation des tâches de déplacement ne peut pas dépasser le nombre d'agents capables –soit 10 : 5 actionneurs et 5 plates-formes polyvalents–, et les dernières détections sont affectées aux agents dès lors que les premières tâches finissent.

Le même scénario est employé pour tester la deuxième équipe, composée de 15 explorateurs, 5 actionneurs et 15 polyvalents. La figure 5.15 montre les résultats de cette équipe dans le mode d'exploration non-dirigée. Le nombre limité d'actionneurs pousse une fois encore le gestionnaire d'assignation à retirer des agents de la tâche d'exploration pour les affecter aux tâches de déplacement sur les points d'intérêt. Le mode d'exploration dirigée présente des résultats très similaires (non présentés

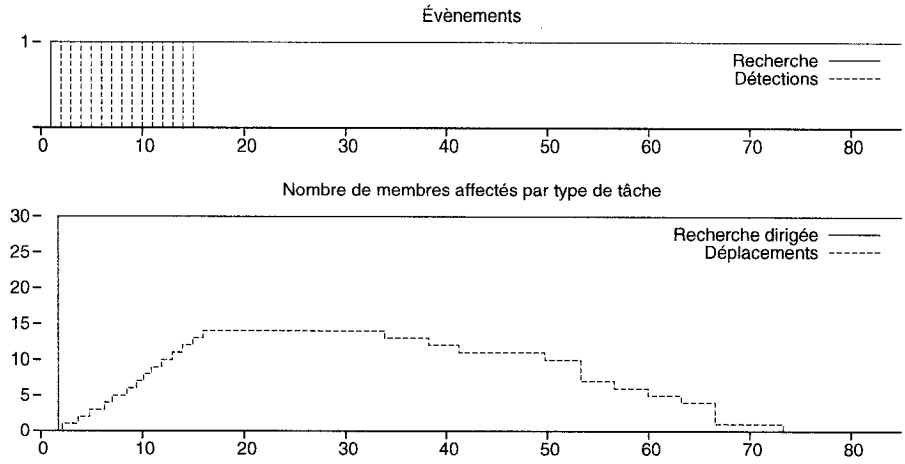


FIG. 5.16 – Résultats d'affectation, exploration dirigée, configuration 15-15-15, scénario 2

ici).

Finalement une dernière équipe est confrontée à ce même scénario : l'équipe est composée de 15 plates-formes de chaque type. La figure 5.16 montre les résultats dans le mode d'exploration dirigée (ici aussi similaire à ceux du mode d'exploration non dirigée). Cette fois, le gestionnaire affecte en tout temps les tâches d'exploration à toutes les plates-formes capables, car le nombre d'actionneurs seuls est suffisant. Le nombre de régions demandées à la segmentation doit être modifié : pour que tous les agents soient affectés, il faut que le nombres de zones d'exploration soient au moins égal au nombre d'agents[4] ; en pratique on choisit une segmentation plus fine (ici 50 zones pour 30 agents). On peut voir alors que l'affectation initiale à la tâche d'exploration est retardée de quelques dixièmes de secondes, représentant la durée de la segmentation et de l'affectation des tâches.

[4]ici, une faille de la définition de mission apparaît : la définition dépend de la composition de l'équipe, ce qu'on cherchait à éviter.

Grâce à l'application présentée au chapitre 4, une validation du fonctionnement de l'ensemble de l'architecture développée dans ce projet a pu être possible. Le système s'est en effet montré capable de prendre en charge diverses équipes de tailles et de compostions différentes pour effectuer les missions proposées.

CONCLUSION

Le projet présenté au cours de ce mémoire avait pour but initial de développer une couche de configuration et de contrôle d'une équipe de plates-formes robotiques. Lors du déploiement de l'architecture associée, de nouveaux défis sont apparus : d'une part, il a fallu adapter et augmenter les outils existants pour le contrôle local de chaque plate-forme, et d'autre part développer une application et les comportements fonctionnels associés pour illustrer les capacités de gestion d'une mission réaliste. Les contributions originales de ce projet sont le développement d'un gestionnaire de mission utilisant une machine à états de type réseau de Pétri dans un système de coordination basée sur une architecture hybride réactive/délibérative pour une équipe hétérogène de plates-formes robotiques mobile, l'intégration de la couche comportementale *Acropolis* développée au *GRPR* à ce système et le développement d'une mission type dans ce contexte.

Coopération et gestion de mission

L'objectif principal était de proposer et développer une architecture permettant d'affecter une mission à une équipe hétérogène de plates-formes robotiques mobiles et de contrôler le déroulement de cette mission dans le temps. Les deux entités logicielles qui ont vu le jour au cours du développement de ce projet, le gestionnaire d'équipe et le proxy d'agent, associés à l'architecture *Acropolis*, dont les fonctionnalités ont été augmentées, ont permis d'atteindre cet objectif principal.

L'ensemble, basé sur le principe d'une architecture réactive/délibérative, permet le contrôle d'une équipe de plates-formes robotiques en respectant le cahier des charges fixé pour le projet :

- gestion d'une équipe de plates-formes aux capacités hétérogènes dont la composition peut varier dans le temps ;
- prise de décision d'équipe et contrôle réactif local ;
- indépendance des différents composants de l'architecture grâce à une architecture en couches ;
- description rapide de missions complexes grâce à un formalisme adapté ;
  développement rapide de comportements pour les plates-formes grâce à la modularité des traitements.

Une utilisation limite de l'architecture pourrait être de contrôler une plate-forme unique, sous la forme d'une équipe comportant un seul membre. Cela permettrait alors de doter sans effort cette plate-forme de missions plus complexes que de simples comportements.

Travail effectué sur le contrôle robotique

Le début du présent projet a coïncidé avec le déploiement au laboratoire de la jeune architecture de prototypage rapide *Acropolis*, dont le développement a été initié par M. Vincent Zalzal. L'étroite relation entre cette architecture et le présent projet m'a naturellement poussé à m'investir dans le développement de cet outil. Au départ du développeur principal, et dans la continuité de ma collaboration, j'ai pris en charge le maintien, la maturation et l'augmentation des fonctionnalités d'*Acropolis*.

L'utilisation de cette architecture dans mon projet, ainsi que dans plusieurs projets connexes au laboratoire, a permis, outre de mettre en lumière l'utilité d'un tel outil, d'en découvrir les limites et de les repousser. Son niveau de maturité permet maintenant d'envisager sa distribution d'une part aux partenaires de recherche

du laboratoire et d'autre part, dans un avenir proche, dans le domaine public sous forme d'un logiciel libre[5]. Une refonte de la structure d'*Acropolis* et de son installation, pour offrir une prise en main plus intuitive aux futurs utilisateurs, a d'ailleurs été un des derniers grands travaux que j'ai menés parallèlement à la finalisation de mon projet.

Complétant l'approche comportementale initialement présente dans la conception d'*Acropolis*, la mise en place de la possibilité de contrôle des comportements par une entité supérieure donne à *Acropolis* de nombreux atouts pour son utilisation dans le développement d'applications robotiques complexes.

Développements futurs

Au cours du présent projet, certaines branches de développement ont dû être écartées pour se focaliser sur l'axe principal afin d'en rendre réaliste l'aboutissement. L'état actuel d'avancement de l'architecture issue de ce projet permet de développer des applications basées sur celle-ci, mais un certain nombre de développements pourrait rendre son utilisation plus attrayante. Deux principales directions devraient pouvoir être prises dans de futures améliorations : rendre plus abordable son approche en proposant des outils d'aide au déploiement des applications et doter l'ensemble de capacités décisionnelles accrues utilisant par exemple des approches de l'intelligence artificielle traditionnelle. Ces développements pourraient affecter les différentes couches de l'architecture : une liste non exhaustive classifie les améliorations possibles selon les modules de l'architecture dans lesquels elles s'inscrivent.

[5]sous licence *GPL : GNU General Public Licence*

Gestionnaire de mission :

- développement d'un outil graphique de conception de missions ;
- développement d'un outil graphique de surveillance de l'état de la mission ;
- ajout de la paramétrisation du nombre de jetons supprimés et créés par les transitions ;
- développement d'un système de production automatique de missions en fonction d'objectifs (planification de missions) ;
- ajout de points de sauvegarde de l'état de la mission, pour reprise ultérieure ou prise en charge par un autre gestionnaire.

Gestionnaire d'équipe :

- développement d'un outil graphique de surveillance de la composition et de l'affectation des membres ;
- ajout de la possibilité pour les agents de connaître leurs équipiers (nombre, noms, capacités, affectation...) pour adapter leurs comportements (par exemple pour régler le problème de la configuration de la segmentation rencontré en 5.3.2).

Proxy d'agent :

- utilisation d'une définition de tâches non mono-comportementales pour le contrôle local : séquence comportementale, machine à états, planification automatique suivant des objectifs, etc... et le développement d'outils d'aide à la conception de ces tâches ;
- intégration du proxy à *Acropolis* sous la forme de modules pour éviter la multiplication des applications nécessaires au démarrage.

***Acropolis* :**

- réécriture de l'outil graphique de création de circuits de configuration ;

- développement d'un outil graphique de surveillance et de journalisation pour aider au développement des comportements.

D'autres développements impliquant l'ensemble de la couche décisionnelle peuvent aussi être envisagés, comme permettre la coexistence de plusieurs gestionnaires redondants pour être plus tolérant aux fautes ou permettre la fusion de plusieurs équipes et de leurs missions respectives.

BIBLIOGRAPHIE

ALBUS, J. ET RIPPEY, W. (1994). RCS : A reference model architecture for intelligent control. In *From Perception to Action Conference, 1994., Proceedings*, pages 218–229.

ARKIN, R. C. (1989). Motor Schema – Based Mobile Robot Navigation. *The International Journal of Robotics Research*, **8**(4), 92–112.

ARKIN, R. C. (1998). *Behavior-based Robotics*. MIT Press, Cambridge, MA, USA.

ARKIN, R. C. ET BALCH, T. R. (1997). AuRA : Principles and practice in review. *Journal of Experimental and Theoretical Artificial Intelligence(JETAI)*, **Volume 9**(Number 2/3), 175–188.

BOTELHO, S. ET ALAMI, R. (1999). M+ : a scheme for multi-robot cooperation through negotiated task allocation and achievement. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1234–1239 vol.2.

BOURGEOIS, F. ET LASSALLE, J.-C. (1971). An extension of the munkres algorithm for the assignment problem to rectangular matrices. *Commun. ACM*, **14**(12), 802–804.

BROOKS, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, **2**(1), 14–23.

BRUNO, J., E. G. COFFMAN, J., ET SETHI, R. (1974). Scheduling independent tasks to reduce mean finishing time. *Commun. ACM*, **17**(7), 382–387.

BUTLER, Z., RIZZI, A., ET HOLLIS, R. (2000). Cooperative coverage of rectilinear environments. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 3, pages 2722–2727 vol.3.

CHAIMOWICZ, L., SUGAR, T., KUMAR, V., ET CAMPOS, M. (2001). An architecture for tightly coupled multi-robot cooperation. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2992–2997 vol.3.

DAVID, R. ET ALLA, H. (1997). *Du grafcet aux reseaux de Petri.* Hermes - Lavoisier, Paris, France.

DIJKSTRA, E. W. (1971). A short introduction to the art of programming. circulated privately.

FARINELLI, A., IOCCHI, L., ET NARDI, D. (2004). Multirobot systems : A classification focused on coordination.

FIKES, R. E. ET NILSSON, N. J. (1971). Strips : A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, **2**, 189–208.

GAT, E. (1991). *Reliable goal-directed reactive control of autonomous mobile robots.* PhD thesis, Blacksburg, VA, USA.

GEORGEFF, M. P. ET LANSKY, A. L. (1987). Reactive reasoning and planning. In *AAAI-87 Proceedings*, pages 677–682. American Association of Artificial Intelligence.

GERKEY, B., VAUGHAN, R., ET HOWARD, A. (2003). The Player/Stage Project : Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323.

GERKEY, B. P. ET MATARIĆ, M. J. (2002). Sold! : Auction methods for multi-robot coordination. *IEEE Transactions on Robotics and Automation*, **18**(5), 758–768.

GERKEY, B. P. ET MATARIĆ, M. J. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, **23**(9), 939–954.

HOWARD, A., SIDDIQI, S., ET SUKHATME, G. (2003). An experimental study of localization using wireless ethernet. In *Proceedings of the International Conference on Field and Service Robotics*.

HUNTSBERGER, T., PIRJANIAN, P., TREBI-OLLENNU, A., DAS NAYAR, H., AGHAZARIAN, H., GANINO, A., GARRETT, M., JOSHI, S., ET SCHENKER, P. (2003). Campout : a control architecture for tightly coupled coordination of multirobot systems for planetary surface exploration. *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, **33**(5), 550–559.

KUBE, C., ZHANG, H., ET WANG, X. (1993). Controlling collective tasks with an aln. In *Intelligent Robots and Systems '93, IROS '93. Proceedings of the 1993 IEEE/RSJ International Conference on*, volume 1, pages 289–293 vol.1.

KUHN, H. W. (1955). The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, **2**, 83–97.

LATIMER, D., I., SRINIVASA, S., LEE-SHUE, V., SONNE, S., CHOSET, H., ET HURST, A. (2002). Towards sensor based coverage with robot teams. In *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, volume 1, pages 961–967 vol.1.

LU, F. ET MILIOS, E. (1997). Robot pose estimation in unknown environments by matching 2d range scans.

LYONS, D. ET HENDRIKS, A. (1992). Planning for reactive robot behavior. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pages 2675–2680 vol.3.

MAES, P. (1990). Situated agents can have goals. In Maes, P., editor, *Designing Autonomous Agents*, pages 49–70. MIT Press.

MATARIĆ, M. J. ET GOLDBERG, D. (2000). Design and evaluation of robust behavior-based controllers for distributed multi-robot collection tasks.

MEYSTEL, A. (1993). Nested hierarchical control. pages 129–161.

MONDADA, F., GAMBARDELLA, L., FLOREANO, D., NOLFI, S., DENEUBORG, J.-L., ET DORIGO, M. (2005). The cooperation of swarm-bots : physical interactions in collective robotics. *Robotics & Automation Magazine, IEEE*, **12**(2), 21–28.

MURPHY, R. R. (2000). *Introduction to AI Robotics*. MIT Press, Cambridge, MA, USA.

PARKER, L. E. (1998). ALLIANCE : An architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation*.

PIRJANIAN, P. (1999). Behavior coordination mechanisms – state-of-the-art.

REKLEITIS, I., DUDEK, G., ET MILIOS, E. (2000). Multi-robot collaboration for robust exploration. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 4, pages 3164–3169 vol.4.

ROBOCUP (2006). Robocup website. Internet. www.robocup.org.

ROSENBLATT, J. K. (1995). DAMN : A distributed architecture for mobile navigation.

SANTAMARÍA, J. C., BALCH, T., BOONE, G., COLLINS, T., FORBES, H., ET MACKENZIE, D. (1997). Io, ganymede and callisto - a multiagent robot trash-collecting team.

SHEHORY, O. ET KRAUS, S. (1995). Task allocation via coalition formation among autonomous agents. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montréal, Québec, Canada, pages 655–661.

ZALZAL, V. (2005). Localisation mutuelle de plates-formes robotiques mobiles par vision omnidirectionnelle et filtrage de Kalman. Mémoire de maîtrise, École Polytechnique de Montréal, Quebec, Canada.

ZLOT, R., STENTZ, A., DIAS, M., ET THAYER, S. (2002). Multi-robot exploration controlled by a market economy. In *Robotics and Automation, 2002.*

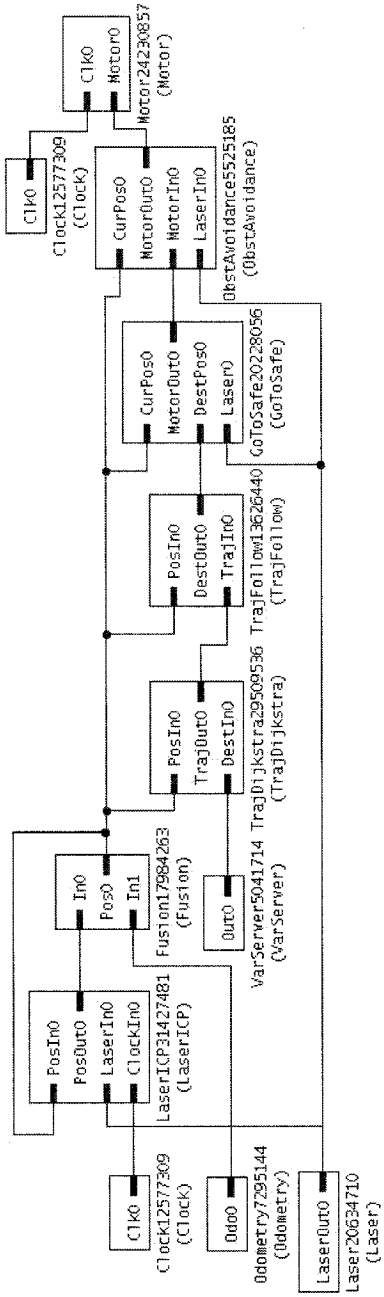
Proceedings. ICRA '02. IEEE International Conference on, volume 3, pages 3016–3023.

ANNEXE I

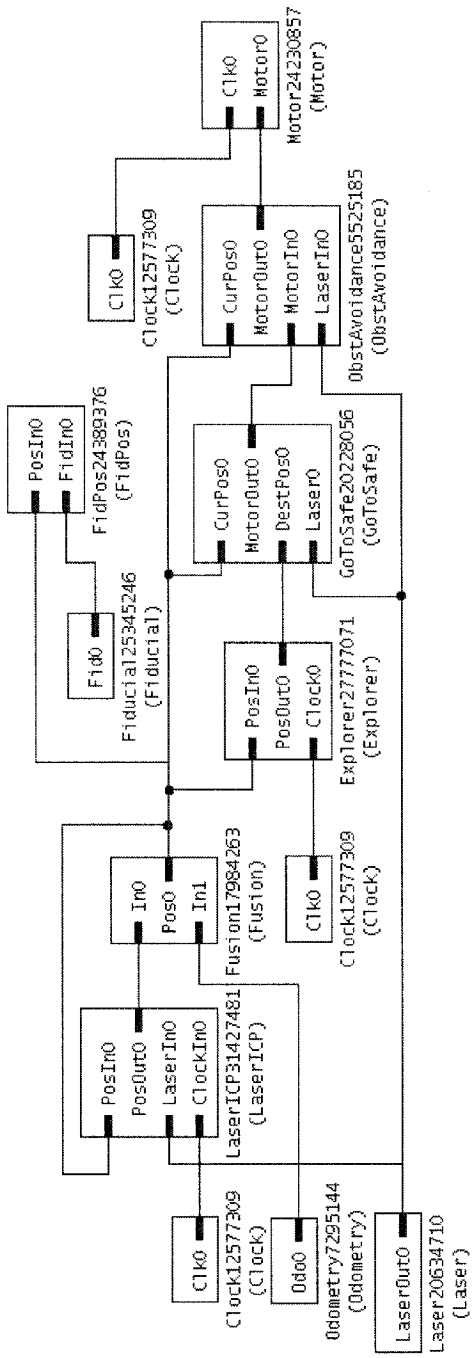
CIRCUITS *ACROPOLIS* UTILISÉS

Cette annexe montre les circuits *Acropolis* qui ont été utilisés pour implanter les différents comportements nécessaires à la mission de *Search-and-Rescue* : le circuit permettant le déplacement sur une carte connue, le circuit d'exploration non dirigée et finalement celui d'exploration dirigée.

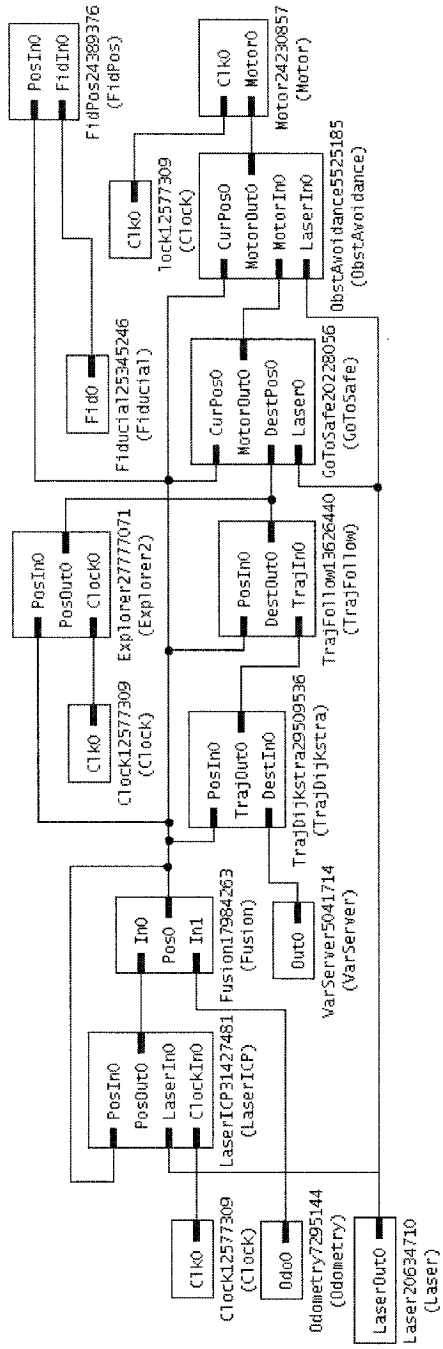
I.1 Circuit *Move*



I.2 Circuit *Search*



I.3 Circuit *DirSearch*



ANNEXE II

CONFIGURATION XML DES RÉSEAUX DE PETRI

Ce document présente la syntaxe du fichier de configuration des réseaux de Pétri associés au gestionnaire de mission. Le fichier de configuration est un fichier XML validé par le descripteur "`SMConfig.dtd`".

Il comporte trois sections regroupées dans une balise *config*, dont les attributs sont la version (*version*, actuellement 1.0) et le nom de la configuration (*name*). La première section (*states*) décrit les places ainsi que les modèles de tâches associés, la seconde (*transitions*) les transitions avec leurs connections aux places et les évènements associés et la dernière (*tokens*) le marquage initial du réseau.

II.1 Balise *states*

```
<states>
 <state name="state_name_1">
  <task type="task_type_1" min="0" max="0">
   <option name="option_name">value</option>
   ...
  </task>
  <task type="task_type_1" min="1" max="2" reassignable="yes"\>
  ...
 </state>
 <state name="state_name_2">
   ...
 </state>
 ...
</states>
```

ALG. II.1 – Section *states*

Dans la section *states* (algo. II.1), chaque place est définie par une balise *state*, avec comme attribut son nom (*name*), qui doit être unique et qui permettra de le référencer dans la définition des transitions et des jetons.

Une place peut avoir un nombre quelconque de modèles de tâche définie par des balises *task*, dont les attributs sont :

- *type* : le type de la tache (obligatoire)
- *min* : le nombre minimum d'agents pour cette tâche (default : *0)*
- *max* : le nombre maximal d'agents pour cette tâche (*0* pour non-limité, default : *0*)
- *reassignable* : *yes/no* la tâche peut etre réassignée (default : *0*)

Chaque tâche peut embarquer des paramètres dans des balises *option* prenant comme attribut son nom (*name*) et comme contenu sa valeur.

II.2 Balise *transitions*

```
<transitions>
 <trans> ... </trans>
 <trans> ... </trans>
 ...
</transitions>
```

ALG. II.2 – Section *transitions*

Dans la section *transitions* (algo. II.2), chaque transition est définie par une balise *trans*, qui accepte comme éléments des balises *from* et *to*, chacune prenant comme attribut le nom de la place (*state)* avec laquelle elle est reliée, et des balises *msg* qui représentent les messages à envoyer lors de l'activation.

II.2.1 Balise *from*

```
<trans>
 <from state="name_input_state_1">
  <event type="event_type" noid="yes"/>
 </from>
 <from state="name_input_state_2" loop="yes"/>
 <from>
   <timer value="5"/>
 </from>
 ...
</trans>
```

ALG. II.3 – Section *transitions* : *from*

Les balises *from* acceptent :

- un argument *loop* (*yes*/*no*), qui définit si le jeton doit etre enlevé de la place d'entrée (default : *no*),
- une balise *event*, représentant l'évènement qui permet l'activation de cette transition (*event* nécessite un attribut *type* pour définir le type d'évènement attendu. *event* accepte un attribut *noid* (*yes*/*no*), qui permet de rendre l'évènement insensible à l'identifiant (default : *no*)),
- une balise *timer*, pour définir une temporisation qui permet l'activation de cette transition (*timer* nécessitte un attribut *value* donnant le nombre de secondes de la temporisation).

II.2.2 Balise *to*

Le seul argument pris par *to* est *state*, qui représente l'état de sortie relié à la transition

```
<trans>
 ...
 <to state="name_output_state_1"/>
 <to state="name_output_state_2"/>
 ...
</trans>
```

ALG. II.4 – Section *transitions* : *to*

II.2.3 Balise *msg*

```
<trans>
 ...
 <msg cmd="command_name_1" plg="plugin_name_1" />
 <msg cmd="vommand_name_2">
  <string>Here is the content of the message</string>
 </msg>
 <msg cmd="command_name_3">
  <optmsg name="name" id="0" type="0" incl_event="no">
   <option name="opt_name_1">option_value</option>
   ...
  </optmsg>
 </msg>
</trans>
```

ALG. II.5 – Section *transitions* : *msg*

msg necessite une commande *cmd* comme argument, et optionnellement l'ensemble des agents (*agent*, default : *) et le plugin (*plg*, default : .) à qui il est destiné. Pour l'agent, * ou *- désigne "pour tous, envoyé avant l'assignation", *+ "pour tous, envoyé après", + "pour les agents assignés par cette transition", - "pour les agents qui sont désassignés", et +- la combinaison des deux. Pour le plugin, on nomme le plugin destinataire, et . représente l'instance de l'agent lui même.

Le contenu du message peut être une chaine de caractères *string*, envoyé directement

(sous la forme d'un *GPlgMsg* pour les plugins et sous la forme d'un *GCtrlMsg* pour les agents), ou un un message à options *GOptionsMsg*, envoyé directement à l'agent, ou sérialisé dans un *GPlgMsg* pour les plugins. Des options statiques *option* peuvent être ajoutées, et les attributs possibles sont :

- *type* : un entier définissant le type de message (default : *0*)
- *name* : le nom associé au message (default : $\emptyset$)
- *id* : l'identifiant de message (default : *0*)
- *incl_tok* : *yes/no* inclut les options des jetons d'entrée (default : *yes*)
- *incl_event* : *yes/no* inclut les options des évènements (default : *yes*)

II.3 Balise *tokens*

```
<tokens>
 <token state="owner_state_name">
  <option name="option_name">value</option>
  ...
 </token>
 <token state="owner_state_name_2"/>
 ...
</tokens>
```

ALG. II.6 – Section *tokens*

La section *tokens* (algo. II.6) représente le marquage initial du réseau. Chaque jeton à insérer est décrit par une balise *token*, qui prend comme attirbut le nom de la place (*state*) dans laquelle il doit être placé, et peut comporter des éléments *option*, comme décrit en II.1 (attributs *name* et contenu).